

Programação C

Ivan L. M. Ricarte

Departamento de Engenharia de Computação e Automação Industrial

Faculdade de Engenharia Elétrica

Universidade Estadual de Campinas

©1995, DCA/FEE/UNICAMP

Conteúdo

1	Introdução	7
1.1	Interação entre Programador e Computador	7
1.2	A Arquitetura de um Computador	8
1.2.1	Representação Numérica	10
1.2.2	Modos de Endereçamento	12
1.3	Sistemas Operacionais	12
1.4	Desenvolvimento de Programas	15
1.4.1	Programação Estruturada	15
1.4.2	Estilo de Codificação	17
1.5	Atividades	18
2	Princípios de Programação C	21
2.1	A História de C	21
2.2	Estruturas de Controle	22
2.3	Expressões	27
2.3.1	Expressões Aritméticas	28
2.3.2	Expressões Lógicas	29
2.3.3	Operadores Binários	30
2.3.4	Operador Condicional	31
2.4	Tipos de Dados	32
2.4.1	Variáveis	33
2.4.2	Arranjos	34
2.4.3	Strings	35
2.4.4	Conversão de Tipos	36
2.5	Estrutura de um Programa C	37
2.5.1	Definição de Funções	38
2.5.2	A Função <code>main</code>	41
2.5.3	A Função <code>printf</code>	42
2.5.4	Escopo de Variáveis	44
2.5.5	Criando um Programa Executável	48
2.6	Atividades	49

3	Apontadores	51
3.1	Variáveis Apontadores	51
3.2	Aritmética de Apontadores	52
3.3	Apontadores e Arranjos	53
3.4	Argumentos na Linha de Comando	55
3.5	Apontador como Argumento de Funções	56
3.6	Apontadores para Funções	58
3.7	Gerenciamento da Memória Livre	60
3.8	O Apontador Nulo	63
3.9	Atividades	64
4	Tipos Derivados	67
4.1	Estruturas	67
4.1.1	Definição de Estruturas	67
4.1.2	Acesso a Elementos	69
4.1.3	Campos	71
4.2	Unões	72
4.3	Enumerações	72
4.4	Definição de Nomes de Tipos	73
4.5	Atividades	75
5	O Pré-processador C	77
5.1	A diretiva <code>#include</code>	77
5.2	A diretiva <code>#define</code>	80
5.3	Compilação Condicional	83
5.4	Outros Recursos	87
5.5	Atividades	89
6	Funções em Bibliotecas	91
6.1	Entrada e Saída de Dados	91
6.1.1	Interação com Dispositivos Padrão	91
6.1.2	Interação com Arquivos	94
6.2	Manipulação de Strings	98
6.3	Interação com o Sistema Operacional	99
6.4	Exemplo de Aplicativo	101
6.5	Atividades	105
7	Introdução a C++	107
7.1	Programação Orientada a Objetos	108
7.2	Particularidades de C++	109
7.2.1	Entrada e Saída	110

7.2.2	Definição de Variáveis	112
7.2.3	Alocação Dinâmica	114
7.2.4	Argumentos de Funções	116
7.3	Classes	118
7.3.1	Construtores e Destrutores	121
7.3.2	Objetos e Funções	122
7.3.3	Sobrecarga de Operadores	122
7.4	Herança	123
7.4.1	Herança de Construtores e Destrutores	125
7.5	Outros Conceitos	126
7.6	Atividades	126
A	Tabela ASCII	127
B	Emacs	129
C	Compilador C	135

Capítulo 1

Introdução

Neste capítulo são introduzidos os conceitos básicos que constituem a base para a codificação em qualquer linguagem de programação. Estes conceitos serão posteriormente revistos sob a óptica da linguagem de programação C.

1.1 Interação entre Programador e Computador

Um computador nada mais faz do que executar instruções que lhe são detalhadamente passadas. Há muitas maneiras de um usuário de um computador requisitar a execução de uma tarefa. Por exemplo, tais interações podem ocorrer através de um comando de um sistema operacional (como `dir` em MS-DOS ou `ls` em Unix para apresentar o conteúdo de um diretório) ou através de interação com uma interface gráfica (por exemplo, carregar um ícone de um arquivo para um símbolo de uma lata de lixo para remover o arquivo do sistema, como em um sistema NeXT).

Estas formas de interação descritas acima representam apenas distintas interfaces para a execução de *programas*. Um programa nada mais é do que uma sequência de instruções para um computador, onde esta sequência de instruções foi definida por um programador. O programador determina as instruções que devem ser executadas em termos de uma *linguagem de alto nível*. Em linguagens de alto nível, as instruções são passadas para o computador usando termos que se aproximam da linguagem humana, de forma a facilitar a expressão e compreensão das tarefas a executar.

Há diversas linguagens de alto nível disponíveis para diversas máquinas distintas. Em geral, cada linguagem teve uma motivação para ser desenvolvida (por exemplo, BASIC para ensinar princípios de programação e Pascal para o ensino de programação estruturada, FORTRAN para programação científica, e lisp para processamento em Inteligência Artificial). A linguagem C é uma destas

linguagens, que foi desenvolvida para facilitar o desenvolvimento de software de sistemas (por exemplos, sistemas operacionais).

O fato de uma linguagem ter sido desenvolvida com uma aplicação em mente não significa que ela não seja adequada para outras aplicações. Este foi o caso com a linguagem C que, juntamente com sua sucessora C++, é atualmente utilizada para um universo muito amplo de aplicações. Uma das grandes vantagens da linguagem C é sua flexibilidade: um programador C tem à sua disposição comandos que permitem desenvolver programas com características de alto nível e ao mesmo tempo trabalhar em um nível muito próximo da arquitetura da máquina, de forma a explorar os recursos disponíveis de forma mais eficiente. Por este motivo, o número de aplicações desenvolvidas em C e C++ (assim como o número de programadores) vêm se ampliando muito em anos recentes.

1.2 A Arquitetura de um Computador

Apesar da evolução tecnológica que ampliou sensivelmente a capacidade de processamento de computadores em anos recentes, a base da organização interna da maior parte dos computadores permanece inalterada. Esta organização, conhecida com a *Arquitetura von Neumann*¹ é diagramaticamente apresentada na Figura 1.1.

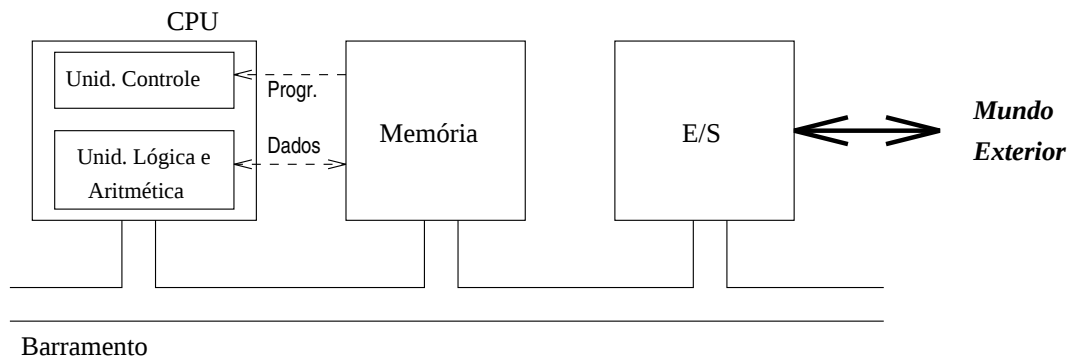


Figura 1.1: Arquitetura von Neumann.

Por esta figura, observa-se que um computador tem três elementos básicos:

processador central (CPU): é o cérebro do computador, que decifra instruções recebidas e as executa;

¹De John von Neumann, matemático húngaro (1903–1957). Atuou como consultor no Projeto ENIAC, em meados da década de 1940.

memória: permite armazenar instruções e dados;

sistema de entrada e saída (E/S): estabelece a comunicação entre o computador e o mundo exterior.

Qualquer interação entre uma aplicação e o computador deve agir sobre um destes três elementos. Cada processador entende uma linguagem própria, que reflete as instruções básicas que ele pode entender. Uma sequência de comandos simbólicos utilizando este nível de instruções é um programa *assembly*, que pode ser convertido para os bits que serão entendidos pelo processador (a *linguagem de máquina*).

Na arquitetura von Neumann, um programa (convertido para a linguagem da máquina) é armazenado na memória. A memória é acessada pelo processador em unidades chamadas *palavras*. A forma de um processador acessar uma palavra específica dentro da memória é através da posição da palavra dentro da memória; esta posição é chamada de *endereço*. Assim, a memória pode ser visualizada como uma sequência de palavras ordenadas segundo seus endereços.

A execução de uma instrução de um programa pode ser vista (simplificadamente) da seguinte forma:

1. o processador busca a instrução do programa;
2. o processador decodifica a instrução, isto é, descobre que ação ele deve realizar para aquela instrução;
3. se esta ação requer dados adicionais que estão em memória, o processador busca também estes dados;
4. o processador executa a ação associada à instrução;
5. o processador armazena o resultado gerado na memória.

Este processo é repetido desde a primeira até a última instrução de um programa.

Quando uma instrução requer uma interação com algum dispositivo externo (por exemplo, entrada via teclado, apresentação de um caracter na tela, acesso a um dado em disco), os dispositivos de entrada e saída são ativados de forma que ocorre uma interação entre estes dispositivos e a memória do computador. Quando a operação é de saída de dados, palavras são transferidas da memória para o dispositivo externo. Quando a operação é de entrada, dados são transferidos do dispositivo externo para a memória do computador.

1.2.1 Representação Numérica

Internamente, todos os dados são representados no computador como sequências de bits². Esta é a forma mais conveniente para manipular os dados através de circuitos digitais, que podem diferenciar apenas entre dois estados (*on* ou *off*, *verdade* ou *falso*, 0 ou 1). Uma sequência de N bits pode representar uma faixa com 2^N valores distintos.

O formato de representação interna (ou seja, como uma sequência de bits é traduzida para um valor) pode variar de computador para computador, embora a busca por uma uniformização para a representação de tipos básicos venha ocorrendo nos últimos anos. Assim, um caracter usualmente ocupa um byte com conteúdo definido pelo código ASCII; um número inteiro tem uma representação binária, em geral em complemento de dois; e um valor real é usualmente representado no padrão IEEE de representação em ponto flutuante.

Inteiros sem sinal (*unsigned*) têm uma representação computacional (em números binários) equivalente à representação usual para números decimais, ou seja, através da atribuição de pesos associados à posição de cada bit. Grande parte dos computadores atuais utilizam 32 bits para representar números inteiros, o que permite representar 4.924.967.296 valores distintos. (A geração mais recente de computadores suporta também inteiros com 64 bits.) Uma sequência binária

$$s_{n-1}s_{n-2}s_{n-3} \dots s_2s_1s_0$$

está associada ao valor inteiro

$$\sum_{i=0}^{n-1} s_i \cdot 2^i$$

onde $s_i \in \{0, 1\}$. O bit s_{n-1} é chamado *bit mais significativo*, enquanto que s_0 é o *bit menos significativo*.

A representação de inteiros com sinal pode usar outros formatos. A forma mais básica é a representação em sinal e magnitude, onde o bit mais significativo denota o sinal associado ao restante da sequência ($s_{n-1} = 1$ indicaria que o número é negativo). Este formato tem a desvantagem de ter duas representações diferentes para o valor zero, além de ter circuitos complicados para suportar operações básicas.

Outra formato suportado para representar inteiros com sinal é a representação em complemento de um. A representação para um número negativo neste formato pode ser obtida facilmente a partir da representação do número positivo correspondente simplesmente complementando cada bit da sequência, ou seja, trocando 0's por 1's e 1's por 0's. Apesar de simplificar circuitos para operações

²Um *bit* é um dígito binário (*binary digit*), usualmente representado pelos símbolos 0 e 1.

básicas, este formato ainda mantém duas representações distintas para o valor zero.

O formato mais aceito para inteiros com sinal é sem dúvida a representação em complemento de dois. Para obter a representação de um número negativo neste formato, computa-se inicialmente a representação em complemento de um e adiciona-se 1 ao bit menos significativo. Neste caso, o valor inteiro associado à sequência $s_{n-1} \dots s_0$ é

$$\sum_{i=0}^{n-2} s_i \cdot 2^i - s_{n-1} \cdot 2^n.$$

Este formato mantém a simplicidade dos circuitos aritméticos e tem apenas uma representação para o valor zero. Uma característica que lhe é peculiar é o fato de que a faixa de valores representáveis não é simétrica em torno de 0, havendo um valor negativo a mais que a quantidade de valores positivos distintos. Por exemplo, sequências de cinco bits podem representar valores entre -16 (10000) e +15 (01111).

Nem todos valores armazenados em variáveis do computador representam números inteiros. O tipo não-numérico mais suportado em diversas linguagens de programação é o caracter (*character*). Sequências de caracteres ocorrem tão frequentemente em computação que recebem um nome específico, (*string*).

Como uma sequência de bits é traduzida em termos de caracteres tem sido o objeto de diversos esforços de padronização, tais como ASCII, ISO-8859, Uni-Code e o *Basic Multilingual Plan* (BMP). O padrão mais aceito é sem dúvida o formato ASCII (*American Standard for Computer Information Exchange*). O formato ASCII básico permite representar 128 caracteres distintos, entre os quais estão diversos caracteres de controle (tais com ESC, associado à tecla de *escape*, e CR, associado ao *carriage return*) e caracteres de pontuação. Neste formato, os caracteres entre '0' e '9' são representados pelos valores hexadecimais 30_H a 39_H , respectivamente (valores decimais entre 48 e 57); as letras maiúsculas 'A' a 'Z', pelos valores de 41_H a $5A_H$; e as letras minúsculas 'a' a 'z', pelos valores de 61_H a $7A_H$.

Outra forma de representação não-inteira corresponde à representação em *ponto flutuante*. Neste formato, associado ao conceito de notação científica, cada número (real) é representado por um sinal, uma mantissa e um expoente. Entre as inúmeras combinações possíveis de formatos de representação que seguem esta filosofia básica, o padrão IEEE-754 tem sido o mais aceito e usualmente suportado em hardware (através das unidades de ponto flutuante em co-processadores ou incorporados a CPUs). Este formato suporta representações de números reais em *precisão simples* (32 bits, dos quais 8 para a representação do expoente e 23 para a representação da mantissa) e em *precisão dupla* (64 bits, sendo 11 para o expoente e 53 para a mantissa). Há também representações especiais para os

valores $-\infty$, $+\infty$ e NaN (*Not a Number*, associado ao resultado de operações sem significado matemático, tal como a divisão de zero por zero).

1.2.2 Modos de Endereçamento

Variáveis de um programa de computador são armazenadas em memória, e referências a estas variáveis devem ser traduzidas internamente em termos de seus *endereços*. Há diversas maneiras possíveis que computadores podem usar para especificar endereços em suas instruções; estas formas são os chamados *modos de endereçamento*. Embora o número de combinações possíveis para a especificação de endereços possa ser praticamente sem fim, processadores mais recentes (da linha RISC³) suportam apenas as formas mais básicas de endereçamento, descritas a seguir.

No modo de endereçamento *direto* o endereço de uma variável é especificado diretamente na instrução. Esta é a forma mais intuitiva de endereçamento.

No modo de endereçamento *indexado*, o endereço é computado a partir da soma de um *endereço base* (mantido em um registro da CPU) e um *deslocamento*. Esta forma de endereçamento é perfeitamente adequada para acesso a sequências de valores, tais como suportadas por vetores numéricos ou strings de caracteres.

O terceiro modo básico é o endereçamento *indireto*, onde o endereço especificado em uma instrução não é o endereço da variável mas sim um endereço onde o endereço da variável poderá ser obtido. Este modo de endereçamento permite suportar um mecanismo de acesso dinâmico, ou seja, uma mesma instrução pode operar sobre variáveis distintas. Em termos de linguagem de programação, este modo de endereçamento está usualmente associado ao conceito de apontadores.

1.3 Sistemas Operacionais

O *Sistema Operacional* é um programa supervisor que estabelece uma camada entre o hardware do computador e aplicações de usuários. Uma de suas funções é estabelecer uma interface de software uniforme entre o computador e outros programas do sistema e programas de aplicação de usuários. Outra função fundamental de um sistema operacional é gerenciar os recursos de um computador de forma a promover sua eficiente utilização. Exemplos de sistemas operacionais são MS-DOS, OS/2 e Linux para computadores pessoais e sistemas Unix para estações de trabalho.

A unidade básica de computação gerenciada por um sistema operacional é um *processo*, que é basicamente um programa em execução. Os recursos requeridos

³*Reduced Instruction Set Computers*, ou seja, computadores com um pequeno número de instruções.

por um processo, tais como espaço em memória para guardar os dados e as instruções do programa, são alocados pelo sistema operacional. Um processo passa a existir a partir do momento em que um programa foi ativado, e deixa de existir quando a execução do programa é encerrada. Neste ponto, os recursos que lhe foram alocados pelo sistema operacional são liberados.

Outra função associada ao sistema operacional é o gerenciamento dos recursos de memória do computador. Em um sistema onde mais de um programa pode estar sendo executado simultaneamente, os diversos processos estão competindo pelos recursos de memória, que são limitados. É tarefa do sistema operacional coordenar estes recursos de forma que, em média, todos os processos tenham tempo de execução em limites razoáveis.

Do ponto de vista lógico, a área de memória de um processo é dividida em segmentos independentes. No sistema operacional Unix, por exemplo, há três segmentos de memória associados a um processo: texto, dados e pilha.

O *segmento de texto* contém as instruções do programa, e geralmente é uma área protegida pelo sistema operacional (isto é, um programa de um usuário não pode modificar diretamente o conteúdo ou o tamanho desta área). Uma tentativa de modificação em uma palavra desta área irá causar um erro de execução de um programa, tal com **segmentation fault**. Deve ficar claro que o conteúdo desta área não é o texto das instruções em uma linguagem de alto nível (C, por exemplo), mas sim sua tradução em termos de linguagem de máquina. Esta tradução é realizada pelo programa *compilador*, um dos aplicativos em um sistema operacional.

O *segmento de dados* armazena dados e variáveis do programa. Esta é uma área cujo conteúdo pode ser modificado (o valor de variáveis) e cujo tamanho pode ser alterado. O segmento de dados pode ser conceitualmente dividido em duas sub-áreas, uma área *estática*, com dimensão conhecida no momento da compilação de um programa, e uma área dinâmica ou *livre*, que pode crescer ou encolher durante a execução do programa. Entretanto, deve-se observar que este tamanho não pode crescer indefinidamente, havendo limites para a memória total associada a um processo.

O *segmento de pilha* mantém as informações temporárias de um processo. Este segmento também tem conteúdo e tamanho variáveis, embora também haja limites para a expansão da pilha. O crescimento ilimitado da pilha pode causar um erro de execução tal como **stack overflow**.

Outro recurso gerenciado pelo sistema operacional é o *sistema de arquivos*, que permite armazenar dados e acessá-los de discos sob a forma de arquivos. Esta é uma atividade importante do sistema operacional, uma vez que o acesso a discos magnéticos pode se tornar um ponto crítico no desempenho de aplicações manipulando muitos dados.

Uma forma que o sistema operacional suporta para melhorar a eficiência de acesso a arquivos é manter, em memória principal, parte da informação sobre

arquivos em disco. Para tanto, o sistema operacional mantém uma *tabela de arquivos*. Esta tabela mantém informações que permitem agilizar o acesso aos dados do arquivo, além de manter informações de suporte (por exemplo, indicação de qual a posição corrente dentro do arquivo). Uma aplicação pode referenciar um arquivo presente nesta tabela através de um *descriptor de arquivo*, um número inteiro indicando uma posição na tabela alocada pelo sistema operacional.

O sistema operacional suporta rotinas para requisitar a abertura de um arquivo (alocando-lhe uma entrada na tabela de arquivos), para fechar um arquivo aberto (liberando espaço nesta tabela), para transferir dados entre o arquivo e a memória principal, e para rotinas de suporte à manipulação de arquivos.

Em Unix, a manipulação de dispositivos de entrada e saída (impressoras, teclados, monitores) é tratada uniformemente sob a forma de arquivos. Por exemplo, o teclado está associado a um arquivo padrão de entrada, enquanto que o monitor está associado ao arquivo padrão de saída e ao arquivo padrão de mensagens de erro.

A Figura 1.2 apresenta o papel do sistema operacional como interface entre aplicativos e o hardware. Aplicativos podem ser tanto os programas desenvolvidos pelo usuário como outras aplicações que suportam ou facilitam o desenvolvimento das aplicações do usuário.



Figura 1.2: Papel do Sistema Operacional.

O objetivo deste curso é fornecer a base de programação em linguagem C necessária para o desenvolvimento de aplicações do usuário, que podem ser auto-contidos, usar recursos do sistema operacional ou recursos de outros aplicativos. A seguir, alguns aspectos ligados ao desenvolvimento deste tipo de aplicações são abordados.

1.4 Desenvolvimento de Programas

Não há dúvidas hoje em dia quanto à importância do software no desenvolvimento dos mais diversos sistemas. Com esta evolução do papel do software em sistemas computacionais, veio também uma maior complexidade de programas e uma maior preocupação em desenvolver programas que pudessem ser facilmente entendidos e modificados (se necessário), não apenas pelo autor do programa mas também por outros programadores. A disciplina que estuda o desenvolvimento de bom software é conhecida como *Engenharia de Software*.

A Engenharia de Software estabelece alguns princípios de desenvolvimento que independem da linguagem de programação adotada. Estes princípios são utilizados nas três fases da vida de um programa, que são:

Definição: estuda o que deve ser feito pelo programa, incluindo análises do sistema e do problema a ser resolvido;

Desenvolvimento: incorpora o projeto do programa (por exemplo, uma descrição do algoritmo e estruturas necessárias), a codificação (tradução do projeto em termos de uma linguagem de programação) e testes do programa; e

Manutenção: é a fase de mudanças decorrentes da correção de erros e atualizações do programa.

A seguir, serão destacados alguns aspectos ligados ao desenvolvimento de software, que será o principal enfoque deste curso.

1.4.1 Programação Estruturada

Programação estruturada estabelece uma disciplina de desenvolvimento de algoritmos que facilita a compreensão de programas. O princípio básico de programação estruturada é que um programa é composto por blocos elementares, onde o fluxo de execução de cada bloco tem um ponto de início (no topo do bloco) e um ponto de término (no fim do bloco). Em outras palavras, na programação estruturada não devem ocorrer “saltos” no fluxo de execução de um programa ligando a parte interna de um bloco a outros segmentos de um programa.

As construções básicas associadas à programação estruturada são sequência, condição e repetição. A seguir, estas construções são detalhadas e possíveis representações em termos de uma pseudo-linguagem de alto nível são apresentadas.

Sequência implementa os passos de processamento necessários para descrever qualquer programa. Por exemplo, um segmento de programa da forma “faça a, depois b e depois c” seria representado por uma sequência

```
a;  
b;  
c;
```

Condição especifica a possibilidade de selecionar o processamento baseado em ocorrências lógicas. Há duas formas básicas de condição. A primeira é a construção *IF-THEN-ELSE*, que permite representar segmentos da forma “se *a* for verdade, faça *b*; senão, faça *c*.” Este segmento poderia ser representado como

```
IF (a)  
  THEN b;  
  ELSE c;  
ENDIF
```

A outra forma de condição é *seleção*, que representa segmentos da forma “se *a* tem o valor *x*, faça *b*; se tem o valor *y*, faça *c*; se tem o valor *z*, faça *d*; e se não tiver nenhum destes valores, faça *e*.” Uma possível representação para tal segmento é

```
CASE OF a:  
  WHEN x SELECT b;  
  WHEN y SELECT c;  
  WHEN z SELECT d;  
  DEFAULT: e;  
ENDCASE
```

Observe que este comando não é essencial, uma vez que a condição de seleção por ele indicada poderia ser representada em termos de IF-THEN-ELSE, como em

```
IF (a = x)  
  THEN b;  
  ELSE IF (a = y)  
    THEN c;  
    ELSE IF (a = z)  
      THEN d;  
      ELSE e;  
    ENDIF  
  ENDIF  
ENDIF
```

O suporte a estruturas do tipo CASE facilitam a expressão de condições que ocorrem frequentemente em programas (por exemplo, selecionar ações dependendo de uma opção escolhida em um menu) sem ter que recorrer ao aninhamento excessivo de condições do tipo IF-THEN-ELSE.

Repetição permite a execução repetitiva de segmentos do programa. Na forma básica de repetição, uma condição *a* é testada e então uma tarefa *b* é executada, sendo repetida enquanto *a* for verdade. Esta construção poderia ser representada por

```
WHILE (a)
    b;
ENDWHILE
```

Qualquer programa, independentemente de sua complexidade ou área de aplicação, pode ser desenvolvido usando apenas estes conceitos fundamentais. Entretanto, em alguns casos pode-se violar (de forma controlada) estes princípios como forma de se tratar condições de exceção de forma mais clara e eficiente (por exemplo, na interrupção de uma iteração).

Programação estruturada permite aplicar de forma natural a estratégia de desenvolvimento por refinamentos sucessivos (a abordagem *top-down*). Em uma primeira etapa de desenvolvimento, apenas as linhas gerais de um programa devem ser definidas, constituindo uma descrição com um alto nível de abstração. À medida que o desenvolvimento do programa avança, estas descrições abstratas irão sendo refinadas em etapas sucessivas. Ao final de uma série de refinamentos (tantos quanto forem necessários), obter-se-á uma descrição já próxima das construções da linguagem de programação que será utilizada, podendo-se então passar para a fase de codificação do programa.

1.4.2 Estilo de Codificação

A tradução de uma especificação de um programa para uma linguagem de programação pode resultar em programas incompreensíveis se não houver um cuidado na preservação da informação presente. As linhas gerais que estão a seguir buscam estabelecer uma disciplina de codificação que, se seguida, facilita o entendimento e manutenção de programas.

Código deve ser acima de tudo claro. Os compiladores modernos fazem um ótimo trabalho de otimização de código, de forma que não há necessidade do programador ficar se preocupando em usar pequenos “truques” para economizar algumas instruções no código de máquina — provavelmente o compilador já faria isto para o programador. A preocupação com a otimização de código só deve existir após a detecção da necessidade real desta otimização, e ela deve se restringir ao segmento do programa onde o problema de desempenho foi localizado.

Nomes de identificadores (variáveis, procedimentos) devem denotar claramente o significado do identificador. Linguagens de programação modernas raramente limitam o número de caracteres que um identificador pode ter, de forma que não

faz sentido restringir este tamanho. Compare as duas linhas de código a seguir, e observe como nomes claros podem facilitar a compreensão de um programa:

```
d = v*t;  
distancia = velocidade * tempo;
```

O uso de comentários deve ser usado como forma de documentação interna de um programa. O excesso de comentários não é recomendado, pois isto pode prejudicar a leitura de um programa. Entretanto, há comentários que devem estar presentes em qualquer programa, tais como

- Comentários de prólogo, que aparecem no início de cada módulo. Devem indicar a finalidade do módulo, uma história de desenvolvimento (autor e data de criação e modificações), e uma descrição das variáveis globais (se houver);
- Comentários de procedimento, que aparecem antes de cada função indicando seu propósito, uma descrição dos argumentos e valores de retorno;
- Comentários descritivos, que descrevem blocos de código.

Comentários devem ser facilmente diferenciáveis de código, seja através do uso de linhas em branco, seja através de tabulações. É importante que comentários sejam corretos, uma vez que um comentário errôneo pode dificultar mais ainda o entendimento de um programa do que se não houvesse comentário nenhum.

Na especificação das construções estruturadas, use tabulações para indicar blocos de distintos níveis. Evite sempre que possível o uso de condições de teste complicadas e o aninhamento muito profundo de condições de teste. Use parênteses para deixar claro como expressões lógicas e aritméticas serão computadas.

Com relação a instruções de entrada e saída, todos os dados de entrada devem ser validados. O formato de entrada deve ser tão simples quanto possível e, no caso de entrada interativa, o usuário deve receber uma indicação de que o programa está aguardando dados. Da mesma forma, a apresentação de resultados deve ser clara, com indicação sobre o que está sendo apresentado. Finalmente, teste o valor de retorno de qualquer rotina que possa retornar um erro, tomando as ações que forem necessárias para minimizar o efeito do erro.

1.5 Atividades

Atividade 1 Apresente a representação binária em complemento de dois (8 bits) para os seguintes valores decimais inteiros:

- (a) 25
- (b) -31
- (c) 74
- (d) -127

Atividade 2 Apresente o valor decimal associado aos seguintes inteiros em representação binária em complemento de dois:

- (a) 10110110
- (b) 01011101
- (c) 00011111
- (d) 11111111

Atividade 3 Desenvolva algoritmos em pseudo-linguagem (comandos em português combinados com estruturas de controle, dadas na Seção 1.4.1) para os seguintes programas. Indique quais parâmetros são necessários, diferenciando entre aqueles que são constantes e aqueles que são variáveis. Assuma a existência de operadores e procedimentos básicos, indicando quando for o caso quais parâmetros de entrada e os resultados dos procedimentos.

- (a) Crie um programa de conversão de valores em reais para dólares.
- (b) Modifique o programa anterior de forma a criar uma tabela de conversão de reais para dólares.
- (c) Generalize os dois programas acima de forma que a conversão inversa (dólares para reais) também seja suportada.
- (d) Crie um programa que, dado o nome de um arquivo, o número de caracteres e o número de linhas no arquivo seja apresentado.
- (e) Crie um programa que verifique se um arquivo contém apenas caracteres ASCII imprimíveis ou não.
- (f) Modifique o programa acima para criar um programa que apresente na tela o conteúdo ASCII de um arquivo, deixando de apresentar os caracteres não-imprimíveis. Ao final da apresentação, o programa deve apresentar um sumário com a contagem de quantos caracteres deixaram de ser apresentados.

Atividade 4 Investigue no manual do seu sistema operacional (por exemplo, a seção 2 do `man` em Unix) que tipo de atividades são suportadas na interface de serviços para programas (os *system calls*). Selecione dez serviços e classifique-os de acordo com o recurso do computador que estará sendo acessado.

Capítulo 2

Princípios de Programação C

2.1 A História de C

A linguagem de programação C foi desenvolvida no início dos anos 70 nos Laboratórios AT&T Bell, nos Estados Unidos. A motivação para que o autor de C, Dennis Ritchie, criasse uma nova linguagem de programação foi o desenvolvimento do sistema operacional Unix. C é uma ferramenta tão básica que praticamente todas as ferramentas suportadas por Unix e o próprio sistema operacional foram desenvolvidas em C.

C acompanhou o ritmo da distribuição do sistema operacional Unix, que foi amplamente divulgado e livremente distribuído na década de 70. Apesar de haver compiladores para linguagens mais “tradicionais” na distribuição Unix, aos poucos C foi ganhando simpatizantes e adeptos. Atualmente, não há dúvidas de que C é uma das linguagens de programação de maior aceitação para uma ampla classe de aplicações.

Um dos grandes atrativos da linguagem C é o balanço atingido entre características próximas da arquitetura de computadores e características de linguagens de programação com alto nível de abstração. O ascendente mais remoto de C, Algol 60, desenvolvida por um comitê internacional, foi uma linguagem que buscava um alto grau de abstração, com estruturas modulares e sintaxe regular. Por Algol ser “abstrata demais”, variantes surgiram que buscavam aproximar aquela linguagem um pouco mais da máquina, tais como CPL (*Combined Programming Language*), desenvolvida na Inglaterra. Esta linguagem era ainda muito complexa, o que dificultava seu aprendizado e a implementação de bons compiladores. BCPL (*Basic CPL*) buscava capturar apenas as características principais de CPL, e B (desenvolvida por Ken Thompson nos Laboratórios Bell, em 1970) levava este objetivo ainda mais adiante. Entretanto, estas linguagens ficaram tão “básicas” que tinham pouca aplicação direta. Ritchie reincorporou algumas características

de alto nível à B, tais como suporte a tipos de dados, para criar a linguagem C.

A simplicidade de C não restringe, no entanto, a potencialidade de suas aplicações. Blocos desempenhando tarefas muito complexas podem ser criados a partir da combinação de blocos elementares, e este mecanismo de combinação de partes pode se estender por diversos níveis. Esta habilidade de construir aplicações complexas a partir de elementos simples é um dos principais atrativos da linguagem.

O sucesso de C foi tão grande que diversas implementações de compiladores surgiram, sendo que nem todos apresentavam o mesmo comportamento em pontos específicos, devido a características distintas arquiteturas de computadores ou a “extensões” que se incorporavam à linguagem. Para compatibilizar o desenvolvimento de programas em C, o Instituto Norte-Americano de Padrões (ANSI) criou em 1983 um comitê com o objetivo de padronizar a linguagem. O resultado deste trabalho foi publicado em 1990, e foi prontamente adotado como padrão internacional. Além de padronizar aspectos básicos da linguagem, ANSI-C também define um conjunto de rotinas de suporte que, apesar de não ser parte integrante da linguagem, deve ser sempre fornecido pelo compilador.

2.2 Estruturas de Controle

Na Seção 1.4.1, princípios de programação estruturada foram introduzidos, com destaque para as construções que permitem controlar o fluxo de execução de um programa. Sendo C uma linguagem que suporta a programação estruturada, construções equivalentes àquelas devem estar presentes na linguagem.

Um comando da linguagem C é uma expressão delimitada pelo símbolo `;`. Sequências de comandos podem ser agrupadas em blocos delimitados pelos símbolos `{` e `}`.

O controle da forma condição (IF-THEN-ELSE) é suportado em C pelo comando `if`. Seja **A** a condição a ser testada, **B** o comando a ser executado caso a condição **A** seja verdadeira e **C** o comando a ser executado caso a condição **A** seja falsa, na linguagem C:

```
if (A)
    B;
else
    C;
```

Observe que:

1. em C, há diferenças entre letras minúsculas e maiúsculas. Como todos os comandos em C, as palavras chaves deste comando estão em letras minúsculas. Assim, um comando IF (ou If ou iF) não seria entendido pelo compilador;

2. a palavra *then* não faz parte deste comando em C. A forma do comando é simplesmente if-else;
3. apesar de o comando estar apresentado com distintos níveis de tabulação, em C este é apenas um recurso para facilitar a visualização da lógica do programa. O compilador não faria nenhuma diferença entre o comando na forma em que ele foi apresentado acima e a codificação em uma mesma linha,

```
if (A) B; else C;
```

A parte *else* do comando é opcional. Assim, o comando

```
if (A)
    B;
```

estaria denotando “se A for verdade, faça B.” Por outro lado, o comando

```
if (A)
    B;
    C;
```

denotaria “se A for verdade, faça B e, independentemente da condição A, faça C.” Lembre-se que C não faz distinção entre linhas e colunas. Se o desejado neste caso fosse “se A for verdade, faça B e C, ou se A for falso, faça D e E” então a forma correta de codificação em C seria delimitar os blocos através do uso de chaves, como em

```
if (A) {
    B;
    C;
}
else {
    D;
    E;
}
```

No caso de haver mais de um if que possa ser associado a uma cláusula *else*, o bloco if mais próximo é associado ao bloco *else*. Assim, no trecho de código

```
if (A1)
    B;
if (A2)
    C;
else
    D;
```

o bloco correspondente à condição A1 não tem uma cláusula `else` associado; D será executado quando A2 for falso, independente da condição A1. Para executar o comando D apenas quando A1 for falso, chaves deveriam ser introduzidas para clarificar esta intenção, como em

```
if (A1) {  
    B;  
    if (A2)  
        C;  
}  
else  
    D;
```

O comando de controle de seleção (CASE-SELECT) também é suportado em C. Um comando da forma “se o valor da variável A for x faça B, se for y faça C, e em qualquer outro caso faça D” seria representado em C na forma

```
switch (A) {  
    case x: B;  
        break;  
    case y: C;  
        break;  
    default: D;  
}
```

Assim como a condição `else` no comando `if` é opcional, a condição `default` também é opcional para o `switch-case`. Observe que neste caso o comando (ou bloco de comandos) associado a cada condição `case` não é delimitado por chaves, mas sim pela palavra chave `break`. Esta palavra chave é fundamental para que se obtenha a lógica correta para este comando. Por exemplo,

```
switch (A) {  
    case x: B;  
    case y: C;  
    default: D;  
}
```

denotaria “se A for igual a x, faça B, C e D; se for igual a y, faça C e D; e se não for nem x nem y, faça D.” Em alguns casos, este pode ser exatamente o comportamento desejado. Porém, quando não for este o caso, o comando `break` não deve ser esquecido.

Comandos de repetição em C são suportados em três formas distintas. A primeira forma, `while`, é equivalente à forma básica `WHILE`:

```
while (A)
    B;
    C;
```

A expressão acima denota “enquanto a condição A for verdade, execute o comando B. Quando a condição A passar a ser falsa, então execute o comando C.” Assim como para o comando if-else, blocos de comandos que devem ser repetidos precisam ser delimitados por chaves, como em

```
while (A) {
    B;
    C;
}
```

que indica “enquanto A for verdade, faça B e C.”

A segunda forma de comando de repetição em C permite representar mais facilmente situações nas quais o comando a ser repetido deve ser executado pelo menos uma vez antes de que a condição de iteração possa ser avaliada. Usando a forma `while`, tal situação seria representada como

```
B;
while (A)
    B;
    C;
```

Com a segunda forma de comando de repetição de C, `do-while`, este trecho é equivalente a

```
do
    B;
while (A);
    C;
```

A terceira forma associada ao comando de repetição em C facilita a expressão de iterações associadas a contadores. Por exemplo, seja `InicCont` o comando que inicializa o contador, `IncrCont` o comando que avalia o próximo valor do contador, e `CondTerm` a condição que será testada para indicar se a repetição deve ser encerrada ou não, então o comando

```
InicCont;
while (CondTerm) {
    A;
    IncrCont;
}
```

pode ser representado de forma mais compacta usando o comando `for`,

```
for (InicCont; CondTerm; IncrCont)
    A;
```

Como nos outros casos, se `A` fosse substituído por uma sequência de comandos estes comandos deveriam estar delimitados entre chaves.

Qualquer que seja forma usada para indicar o comando de repetição, há duas formas de se desviar a sequência padrão de execução do comando em C. A primeira forma, `continue`, serve para indicar o fim prematuro de *uma* iteração, como em

```
while (A) {
    B;
    if (FimPrematuro)
        continue;
    C;
}
```

Neste caso, se a condição `FimPrematuro` for verdadeira após a execução de `B`, a iteração é interrompida sem que `C` seja executado, e a condição `A` é imediatamente reavaliada. Se `A` ainda for verdade, então o comando de repetição continuará sendo executado.

A outra forma de interrupção de um comando de repetição é o comando `break`, que indica o fim prematuro de *todo* o comando de iteração, como em

```
while (A) {
    B;
    if (FimPrematuro)
        break;
    C;
}
D;
```

Neste caso, caso a condição `FimPrematuro` seja verdadeira, o comando de repetição é totalmente interrompido, sendo `D` o próximo comando a ser executado.

A linguagem C também suporta a forma `goto`. Por exemplo, a mesma sequência acima poderia ser representada de forma equivalente como

```
while (A) {
    B;
    if (FimPrematuro)
        goto rotulo;
```

```
        C;  
    }  
rotulo:  
    D;
```

Como a forma `goto` pode dificultar a compreensão de programas, ferindo os princípios da programação estruturada, é uma recomendação geralmente aceita entre programadores C que este comando deve ser evitado.

2.3 Expressões

Na Seção 2.2, expressões da linguagem C foram representadas simplesmente por letras A, B, C, ... Nesta seção, o formato destas expressões será apresentado. Serão apresentados os operadores básicos da linguagem, sendo que outros operadores serão introduzidos juntamente com os conceitos que lhe são associados.

Antes de apresentar a forma dos comandos, é interessante que se apresente a forma de se introduzir comentários em um programa C. Comentários desempenham um papel importante em qualquer linguagem de programação, servindo como mecanismo de clarificação de códigos.

Comentários em C são indicados pelos terminadores `/*` (início de comentário) e `*/` (fim de comentário). Quaisquer caracteres entre estes dois pares de símbolos são ignorados pelo compilador. Comentários em C não podem ser aninhados, mas podem se estender por diversas linhas e podem começar em qualquer coluna.

Expressões em C são terminadas pelo símbolo `;` (ponto e vírgula). A expressão mais simples em C é o comando nulo, indicado simplesmente pelo símbolo terminador. Por exemplo,

```
while (Cond)  
    ; /* nao faca nada enquanto Cond for verdade */
```

O comando de atribuição em C é indicado pelo símbolo `=`, como em

```
A = B; /* variavel A recebe valor da variavel B */  
C = 10; /* variavel C recebe valor constante 10 */  
D = 'A'; /* variavel D recebe caracter 'A' */
```

Em C, alguns valores constantes têm representação especial. Caracteres ASCII são denotados entre aspas simples, tais como `'A'` (caracter ASCII A, equivalente ao valor hexadecimal 41_H ou decimal 65). Além dos caracteres alfanuméricos e de pontuação, C também define representações para caracteres especiais através de sequências de escape iniciados pelo símbolo `\` (contrabarra). Tais sequências são:

<code>\n</code>	nova linha
<code>\t</code>	tabulação
<code>\b</code>	retrocesso
<code>\r</code>	retorno de carro
<code>\f</code>	alimentação de formulário
<code>\\</code>	contrabarra
<code>\'</code>	apóstrofo
<code>\"</code>	aspas
<code>\xxx</code>	padrão de bits <code>xxx</code> em octal
<code>\0</code>	o caracter NUL

Além das constantes que requerem uma representação com escape, números inteiros em C podem ser representados em decimal (qualquer sequência de algarismos entre 0 e 9 que inicie com um algarismo diferente de 0), octal (sequências de algarismos entre 0 e 7 iniciada por 0) ou hexadecimal (sequências de algarismos entre 0 e F iniciada pelo prefixo `0x`).

Valores com representação em ponto flutuante (reais) são representados em C através do uso do ponto decimal, como em `1.5` para representar o valor um e meio. A notação exponencial também pode ser usada, como em `1.2345e-6` ou em `0.12E3`.

2.3.1 Expressões Aritméticas

Expressões aritméticas em C podem envolver os operadores binários (isto é, operadores que tomam dois argumentos) de soma (+), subtração (-), multiplicação (*), divisão (/) e resto da divisão inteira ou módulo (%). Valores negativos são indicados pelo operador unário -; observe que não há um operador unário + para denotar valores positivos.

C tem formas não usuais (para outras linguagens de alto nível) de expressar as operações de incremento e decremento através dos operadores ++ e --, respectivamente. Por exemplo,

```
A++;    /* equivale a A = A+1 */
++A;    /* o mesmo */
B--;    /* equivale a B = B-1 */
--B;    /* idem */
```

Cada um destes operadores tem duas formas de uso, dependendo se eles ocorrem antes do nome da variável (pré-incremento ou pré-decremento) ou depois do nome da variável (pós-incremento ou pós-decremento). No caso dos exemplos acima, onde os operadores de incremento e decremento ocorrem de forma isolada em uma expressão, as duas formas possíveis são equivalentes. A diferença entre

eles ocorre quando estes operadores são combinados com outras operações. Por exemplo

```
X = A++; /* equivale a X = A;
          seguido por A = A+1; */
Y = ++B; /* equivale a B = B+1;
          seguido por Y = B; */
```

C tem também uma forma compacta de representar expressões na forma

```
var = var op (expr);
```

onde uma mesma variável `var` aparece nos dois lados de um comando de atribuição. A forma compacta é

```
var op= expr;
```

Por exemplo,

```
A += B; /* equivale a A = A+B */
C *= 2; /* equivale a C = C*2 */
```

2.3.2 Expressões Lógicas

As condições lógicas em C são avaliadas segundo um valor inteiro associado à expressão lógica. Em C, o inteiro 0 representa a condição lógica *falso*. Qualquer valor diferente de 0 equivale ao valor lógico *verdade*. Assim, no trecho de código

```
if (a)
    X;
else
    Y;
```

a expressão representada por `Y` será apenas executada quando o valor de `a` for 0, enquanto que a expressão representada por `X` será executada quando `a` tiver qualquer outro valor (1, -1, 525, etc.).

Há seis operadores relacionais (isto é, operadores que resultam valores lógicos) em C, que são:

```
> maior que
>= maior que ou igual a
< menor que
<= menor que ou igual a
== igual a
!= diferente de
```

Expressões envolvendo estes operadores têm resultado igual a 0 (quando a expressão for falsa) ou 1 (quando for verdadeira). Um exemplo de seu uso é

```
if (x > y)
    while (z != 0)
        A;
else if (x < y)
    do
        B;
        while (z == 0);
else
    C;
```

Além dos operadores lógicos, C também suporta conectores lógicos. O conector `&&` (*and*) resulta em 1 quando as duas expressões envolvidas são verdadeiras (ou diferente de 0). O conector `||` (*or*) resulta em 1 quando pelo menos uma das duas expressões envolvidas é diferente de 0. Além destes dois conectores binários, há também o operador unário de negação, `!`, que resulta em 0 quando a expressão envolvida é verdadeira (diferente de 0) ou resulta em 1 quando a expressão envolvida é falsa (igual a 0). Por exemplo,

```
if (x >= y && w != 0)
    while (z != 0)
        A;
else
    do
        B;
        while (!(z == 0 || w > 0));
```

Expressões lógicas complexas, envolvendo diversos conectores, são avaliadas da esquerda para a direita. Além disto, `&&` tem precedência maior que `||`, e ambos têm precedência menor que os operadores lógicos relacionais e de igualdade. Entretanto, recomenda-se sempre a utilização de parênteses em expressões para tornar claro quais operações são desejadas. A exceção a esta regra ocorre quando um número excessivo de parênteses pode dificultar ainda mais a compreensão da expressão; em tais casos, o uso das regras de precedência da linguagem pode facilitar o entedimento da expressão.

2.3.3 Operadores Binários

A linguagem C oferece também operadores que trabalham sobre a representação binária de números inteiros e caracteres. Estes operadores são:

<code>&</code>	<i>AND</i>
<code> </code>	<i>OR</i>
<code>^</code>	<i>XOR</i>
<code><<</code>	deslocamento à esquerda
<code>>></code>	deslocamento à direita
<code>~</code>	complemento de um

Estes operadores tomam dois argumentos exceto pelo operador `~`, que é unário. Por exemplo,

```
/* a recebe 7 bits menos signif. de x */
a = x & 0177;
/* zera os 8 bits menos signif. de b */
b &= ~0xFF;
/* desloca a represent. de c 4 bits a direita */
c >>= 4;
```

2.3.4 Operador Condicional

A linguagem C oferece um operador ternário que simplifica a representação de construções semelhantes a

```
if (a > b)
    z = a;
else
    z = b;
```

Para usar a forma alternativa, a expressão condicional seria escrita

```
z = (a > b) ? a : b;
```

A forma geral de uso deste operador é

```
expr1 ? expr2 : expr3;
```

Neste comando, a expressão `expr1` é inicialmente avaliada. Caso seu valor seja diferente de 0 (verdade), então a expressão `expr2` é avaliada, tornando-se o valor resultante da expressão condicional. Caso contrário (`expr1` igual a 0), a expressão `expr3` é avaliada e seu resultado é o valor resultante da expressão condicional.

2.4 Tipos de Dados

A linguagem C suporta os tipos de dados básicos usualmente suportados pelos computadores. Na linguagem padrão, os tipos básicos suportados são:

<code>char</code>	caracter
<code>int</code>	inteiro
<code>float</code>	real
<code>double</code>	real de precisão dupla

O tipo `char` ocupa um único byte, sendo adequado para armazenar um caracter do conjunto ASCII.

O tipo `int` representa um valor inteiro que pode ser positivo ou negativo. O número de bytes ocupado por este tipo (e consequentemente a faixa de valores que podem ser representados) refletem o tamanho “natural” do inteiro na máquina onde o programa será executado. Usualmente, quatro bytes são reservados para o tipo `int` nos computadores atuais.

Os tipos `float` e `double` representam valores reais, sendo que `float` oferece cerca de seis dígitos de precisão enquanto que `double` suporta o dobro da precisão de um `float`.

Alguns destes tipos básicos podem ser modificados por *qualificadores*. Por exemplo, o tipo `char` pode ser acompanhado pelo qualificador `signed` ou `unsigned`. O tipo `signed char` seria utilizado para indicar que a variável do tipo `char` estaria representando pequenos números inteiros (na faixa de -128 a 127). O tipo `unsigned char` seria utilizado para indicar que a variável estaria armazenando valores inteiros exclusivamente positivos (sem sinal) na faixa de 0 a 255.

O tipo `int` também pode ser qualificado. Um tipo `unsigned int` indica que a variável apenas armazenará valores positivos. Um tipo `short int` indica que (caso seja possível) o compilador deverá usar um número menor de bytes para representar o valor numérico — usualmente, dois bytes são alocados para este tipo. Uma variável do tipo `long int` indica que a representação mais longa de um inteiro deve ser utilizada, sendo que usualmente quatro bytes são reservados para variáveis deste tipo. Estas dimensões de variáveis denotam apenas uma situação usual definida por boa parte dos compiladores, sendo que não há nenhuma garantia quanto a isto. A única coisa que se pode afirmar com relação à dimensão de inteiros em C é que uma variável do tipo `short int` não terá um número maior de bits em sua representação do que uma variável do tipo `long int`¹.

¹Estes qualificadores refletem o estado atual do padrão ANSI-C. Dependendo do computador e do compilador disponível, outros qualificadores podem ser suportados, tais como `long long` e `long double` para representar inteiros de 64 bits e reais de precisão estendida (80 bits), respectivamente. Entretanto, não há ainda nenhuma padronização neste sentido.

2.4.1 Variáveis

Exemplos válidos de declaração de variáveis em C são:

```
int um_inteiro;
unsigned int outro_inteiro;
char c1, c2;
float SalarioMedio;
double x,
       y;
```

Nomes de variáveis podem ser de qualquer tamanho, sendo que usualmente nomes significativos devem ser utilizados. C faz distinção entre caracteres maiúsculos e caracteres minúsculos, de forma que `SalarioMedio` é diferente de `SalarioMedio`.

Há restrições aos nomes de variáveis. Palavras associadas a comandos e definições da linguagem (tais como `if`, `for` e `int`) são *reservadas*, não podendo ser utilizadas para o nome de variáveis. O nome de uma variável pode conter letras e números, mas deve começar com uma letra.

Como pode ser observado no exemplo acima, diversas variáveis de um mesmo tipo podem ser declaradas em um mesmo comando, sendo que o nome de cada variável neste caso estaria separado por vírgulas. Além disto, variáveis podem ser também inicializadas enquanto declaradas, como em

```
int a = 0,
    b = 20;
char c = 'X';
long int d = 123456L;
```

Observe o uso do sufixo `L` para indicar que uma constante é do tipo `long`.

Uma variável cujo valor não será alterado pelo programa pode ser qualificada como `const`, como em

```
const int NotaMaxima = 100;
```

Neste caso, a variável `NotaMaxima` não poderá ter seu valor alterado (por exemplo, ao estar presente no lado esquerdo de uma atribuição). Evidentemente, variáveis deste tipo devem ser inicializadas no momento de sua declaração.

Outro qualificador ligado à modificação de valor de uma variável é `volatile`, como em

```
volatile int UltimaTecla;
```

Este qualificador indica que a variável poderá ter seu valor modificado por eventos do sistema (por exemplo, o tratamento de uma interrupção ou de um sinal). Isto indica para o compilador que ele não poderá assumir que a variável manterá seu valor simplesmente pela inspeção do código. Caso isto não fosse indicado, o compilador poderia alterar a lógica do programa durante a fase de otimização de código (por exemplo, quando removendo atribuições constantes de dentro de um comando de iteração).

Estes dois qualificadores podem ser combinados: uma variável do tipo `const volatile` seria uma variável cujo valor pode ser alterado *unicamente* por eventos do sistema, e não por comandos de atribuição no programa.

2.4.2 Arranjos

Um *arranjo* é um conjunto de variáveis de um mesmo tipo que é referenciado sob um mesmo nome. O acesso a cada variável de um arranjo é especificado a partir do nome do arranjo e de um *índice* que determina qual elemento deve ser acessado.

Em C, arranjos são definidos e acessados através do operador de indexação `[]`, como em:

```
main() {
    /* declara um arranjo elem de cinco inteiros: */
    int elem[5];
    int i;

    /* inicializa cada elemento do arranjo */
    for (i=0; i<5; ++i)
        elem[i] = 0;

    ...
}
```

Observe através deste exemplo que o primeiro elemento de um arranjo em C é o elemento de índice 0 (`elem[0]`). Consequentemente, para um arranjo com N elementos o último elemento é o de índice N-1 (`elem[4]` no exemplo).

Os elementos de um arranjo podem ser também inicializados durante a declaração da variável. Para o exemplo acima,

```
main() {
    /* declara um arranjo de cinco inteiros */
    int elem[5] = {0,0,0,0,0};
    int i;
```

```
    ...  
}
```

O índice de um arranjo pode ser qualquer expressão inteira, incluindo-se variáveis e constantes inteiras. C não verifica se o valor do índice está dentro da faixa declarada — é responsabilidade do programador garantir que o acesso esteja dentro dos limites de um arranjo.

Arranjos podem ser multidimensionais. Por exemplo, um arranjo bidimensional pode ser declarado, inicializado e acessado como em

```
main() {  
    /* declara um arranjo de duas linhas  
       com tres inteiros por linha */  
    int elem[2][3] = { {0,0,0}, {0,0,0} };  
    int i, j;  
  
    /* modifica conteudo do arranjo */  
    for (i=0; i<2; ++i)  
        for (j=0; j<3; ++j)  
            elem[i][j] = i*3 + j;  
}
```

Em C, um arranjo bidimensional é na verdade um arranjo unidimensional onde cada elemento é um arranjo. Por este motivo, dois operadores de indexação são utilizados ao invés da forma `[i,j]`. Elementos são armazenados por linha.

Em geral, arranjos com muitas dimensões não são utilizados em C visto que ocupam muito espaço e o acesso a seus elementos não ocorre de forma eficiente.

2.4.3 Strings

Um dos tipos de arranjos que mais ocorre em C é o arranjo de caracteres, ou *string*. C não suporta um tipo básico `string`; ao invés, há uma convenção para tratamento de arranjos de caracteres que permite o uso de diversas funções de manipulação de strings na linguagem.

Por convenção, C considera como um string um arranjo de caracteres cujo último elemento é o caracter `NUL`, ou `'\0'`. Por exemplo, um string poderia ser declarado e inicializado como em

```
char exemplo[4] = {'a','b','c','\0'};
```

Observe que o espaço para o caracter `'\0'` deve ser previsto quando dimensionando o tamanho do arranjo de caracteres que será manipulado como string.

C suporta uma forma alternativa de representação de um string constante, que é através do uso de aspas:

```
char exemplo[4] = "abc";
```

Este exemplo é equivalente ao anterior — o string "abc" contém quatro caracteres, sendo que o caracter '\0' é automaticamente anexado ao string pelo compilador.

2.4.4 Conversão de Tipos

A linguagem C é bastante flexível no que se refere a misturar tipos distintos em uma mesma expressão. Por exemplo, ao se atribuir o resultado de uma expressão inteira a uma variável do tipo `float`, o resultado da expressão será automaticamente convertido para aquele tipo.

Apenas conversões que fazem sentido são automaticamente realizadas. Por exemplo, toda variável do tipo `char` que aparece em alguma expressão aritmética é automaticamente convertida para inteiro. Por exemplo, se `c` é uma variável do tipo `char`, então a expressão

```
c - '0';
```

retorna um inteiro. Se `c` tem um caracter que é um dígito — ou seja, está na faixa de valores ASCII entre '0' e '9' — então esta expressão retorna o valor inteiro correspondente ao dígito.

A sequência de conversão automática em expressões aritméticas em C é a seguinte:

- `char` e `short` são convertidos para `int`;
- se um dos operandos é `double`, o outro é convertido para `double` e o resultado da expressão é `double`;
- senão, se um dos operandos é `float`, o outro é convertido para `float` e o resultado da expressão é `float`;
- senão, se um dos operandos é `long`, o outro é convertido para `long` e o resultado da expressão é `long`;
- senão, se um dos operandos é `unsigned`, o outro é convertido para `unsigned` e o resultado da expressão é `unsigned`;
- senão, os operandos são todos `int` e o resultado da expressão é `int`.

No caso de atribuições, o tipo que está à direita do sinal de atribuição é convertido para o tipo da variável que está à esquerda do sinal de atribuição. Quando o tipo à direita é “maior” que o tipo à esquerda, sua representação binária é truncada (ou seja, os bits mais significativos são descartados) para se ajustar ao tipo resultante.

Apesar de conveniente, as conversões automáticas em C podem se tornar a fonte de comportamentos não esperados. Cuidados devem ser tomados quando misturando tipos de dados em expressões e atribuições.

C fornece um mecanismo que permite o programador forçar uma conversão, que é o operador de *molde*. Este operador é da forma (tipo) expressão, como em

```
float pi = 3.1416;
int x;

x = (int) pi;          /* x recebe o valor 3 */
```

Neste exemplo, a expressão (int) pi converte o valor da variável pi para inteiro, resultando em 3. Se x fosse do tipo float, o valor seria então convertido (por atribuição) para 3.0, ou seja, uma variável real com o valor da parte inteira da variável pi.

2.5 Estrutura de um Programa C

Para conhecer um pouco da estrutura de um programa C, um pequeno exemplo será analisado. O programa deste exemplo tem a função de escrever na tela a frase “C ou não C, eis a questão!”. Para tanto, deve-se utilizar um editor de programas de seu sistema (por exemplo, em Unix pode-se utilizar **emacs**, apresentado no Apêndice) para criar um arquivo chamado **ser.c** com o seguinte conteúdo²:

```
main() {
    printf("C ou nao C, eis a questao!\n");
}
```

Para compilar este programa, é necessário invocar o compilador C, como em
\$ cc ser.c

Neste exemplo, o caracter \$ está sendo usado para indicar o *prompt* do sistema operacional. Dependendo da configuração da máquina onde o programa estará sendo desenvolvido, este prompt poderá ter aparência diferente desta.

²A função printf permite imprimir strings e valores na tela, e será vista em mais detalhes na Seção 2.5.3.

O nome de um arquivo fonte com código C deve terminar com `.c`. Caso contrário, o compilador não irá reconhecer que o arquivo contém código C para ser compilado.

Se tudo correu bem, o prompt do sistema deve aparecer novamente. Em Unix, a linha de comando acima gera um arquivo executável de nome `a.out`. Executando este arquivo, a mensagem deve aparecer na tela:

```
$ a.out
C ou nao C, eis a questao!
$
```

Não há nada de especial com relação a este nome `a.out`. Qualquer outro nome poderia ser usado aqui. Se o nome deste arquivo for alterado para qualquer outro nome, ainda será possível executar o mesmo programa. Para usar outro nome, pode-se renomear `a.out` usando o comando do sistema operacional (`mv` em Unix) ou indicando o nome desejado com o uso de uma *chave de compilação*, como em `$ cc ser.c -o ser`

para criar um arquivo executável com o nome `ser`.

Toda a atividade associada a programas C está organizada em forma de *funções*. Um programa C nada mais é do que um conjunto de definições de funções. A seguir, funções C serão analisadas em mais detalhes.

2.5.1 Definição de Funções

Funções contêm declarações e comandos que dizem ao computador o que deve ser feito. Este conjunto de declarações e comandos constituem o *corpo* de uma função.

O corpo da função está delimitado por chaves. Assim como para blocos de comandos, as chaves podem ocorrer em qualquer coluna. O exemplo acima poderia ser reescrito como

```
main()
{
    printf("C ou nao C, eis a questao!\n");
}
```

ou

```
main() { printf("C ou nao C, eis a questao!\n"); }
```

A seleção por um estilo ou outro é uma questão de gosto pessoal. É interessante, no entanto, que um estilo seja adotado e mantido de forma a facilitar a compreensão de código em programas mais extensos.

No caso da função deste exemplo, é interessante observar que

- neste arquivo, a função `main` está sendo *definida*. É possível saber que `main` é uma função por causa dos parênteses depois do identificador. Além do mais, é possível saber que esta função está sendo definida porque seu corpo (sua sequência de código) está presente entre chaves após os parênteses;
- o corpo desta função `main` é composto por uma única linha. Esta linha *invoca* uma outra função (`printf`), que irá efetivamente enviar uma mensagem para a tela. Mais uma vez, é possível saber que `printf` é uma função por causa dos parênteses depois do identificador;
- estas duas funções estão definidas em arquivos diferentes, e foram combinadas pelo compilador para gerar o programa executável.

Em geral, a definição de uma função C tem a forma

```
tipo nome(lista de argumentos) {  
    declaracoes;  
    comandos;  
}
```

O *tipo* indica o valor de retorno de uma função. Todos os tipos que podem ser associados a variáveis em C podem ser utilizados para especificar o valor de retorno da função. O mecanismo de retornar o valor desde uma função é através do comando `return`, como em

```
return(expr);
```

onde `expr` é qualquer expressão cujo resultado esteja de acordo com o tipo de retorno esperado para a função. Se o tipo da função for omitido (como neste exemplo), o compilador irá assumir que o tipo `int` será retornado.

Uma função pode não retornar nenhum valor. Em tais casos, o tipo associado à função é o tipo `void`, como em

```
void vazia() {  
}
```

que é um exemplo de uma função que não faz nada. (Uma função vazia pode ser útil durante algumas fases de desenvolvimento de programas maiores.)

Independente de uma função retornar ou não um valor, um cuidado que deve ser tomado é manter a coerência entre o tipo de retorno esperado e o tipo da expressão que efetivamente estará definindo o valor de retorno. Por exemplo, não retornar nenhum valor quando um `int` é esperado como valor de retorno, como em

```
int func() {  
    /* faz alguma coisa... */  
    return;    /* forma valida de uso - mas nao aqui! */  
}
```

fará com que a rotina que invoca `func` possa receber lixo ao invés de algum valor válido. Por outro lado, o valor de retorno de uma função pode ser ignorado pela função que a chama. Por exemplo, a função `main` do exemplo anterior ignora o valor de retorno de `printf`.

A *lista de argumentos* indica o tipo e o nome que a função dá aos parâmetros que são passados para a função. Argumentos constituem o mecanismo de interface entre funções. Por exemplo, a função

```
int max(int x, int y) {  
    if (x > y)  
        return(x);  
    else  
        return(y);  
}
```

recebe dois inteiros `x`, `y` e retorna o valor do maior dos argumentos. Esta função poderia ser invocada a partir de outra como

```
int a, b, maior;  
...  
maior = max(a, b);
```

Argumentos de funções são passados *por valor*. Em outras palavras, a função `max` invocada acima não acessa diretamente o conteúdo das variáveis `a` e `b`, mas sim cópias temporárias que são copiadas para as variáveis `x` e `y` da função, respectivamente.

Declarações devem estar presentes na primeira parte do corpo, sendo seguidas pelos comandos. Em C, esta ordem deve ser preservada. (Em C++, este não é exatamente o caso.) O segmento de declarações pode incluir:

- definições de variáveis, com ou sem inicialização de valores. Por exemplo,

```
int idade;
```

está definindo uma variável do tipo inteiro de nome `idade`, cujo valor inicial é indefinido. Para indicar que esta variável deve ter o valor inicial igual a zero, a declaração com inicialização

```
int idade=0;
```

poderia ser utilizada. Em C, todas as variáveis devem ser declaradas antes de ser utilizadas;

- declarações de funções que serão utilizadas pela função sendo definida.

A declaração de funções indicam qual o tipo de retorno e quais os tipos dos argumentos que se aguardam em funções que serão invocadas dentro do corpo da função sendo definida. Por exemplo, para indicar que uma função de nome `calcula_idade` será usada na seção de comandos da função sendo definida, e que esta função deve receber três valores inteiros (representando a data de nascimento como dia, mês e ano) e retornará um valor inteiro (a idade calculada para o dia de hoje), a seguinte declaração poderia ser incluída:

```
int calcula_idade(int dia, int mes, int ano);
```

Uma declaração como esta recebe o nome de *protótipo* da função. No protótipo, o nome das variáveis (neste exemplo, `dia`, `mes`, `ano`) são equivalente a comentários, e serve apenas para clarificar o papel associado a cada argumento. Assim, o protótipo

```
int calcula_idade(int, int, int);
```

poderia ser da mesma forma utilizado.

Por fim, os comandos descrevem que ações a função deve desempenhar. Estes comandos são expressões C, possivelmente contendo invocações a outras funções. Se um comando `return` é encontrado, a execução da função é encerrada e o valor de sua expressão (se algum) é retornado para a função que a invocou. Caso nenhum `return` seja encontrado, então todos os comandos até o fim do corpo da função (isto é, até encontrar o último `}` da função) são executados.

2.5.2 A Função `main`

Um programa C é um conjunto de funções, e apenas uma regra existe com relação ao número e ao nome destas funções: deve haver pelo menos uma função, e pelo menos uma função deve ser chamada `main`.

Como já observado, a função `main` é uma função que deve estar sempre presente em um programa C. Esta função determina o escopo de atividade do programa: ela indica qual o ponto de início das atividades do programa (a primeira linha desta função) e qual o ponto de encerramento das atividades do programa (o

último comando da função). O compilador C espera encontrar esta função quando chega à última etapa da compilação, a ligação. Caso a função `main` não seja encontrada, o processamento do compilador será interrompido e uma mensagem de erro será exibida.

A partir de `main`, um número arbitrário de funções pode ser invocado, e estas funções por sua vez podem invocar outras, e assim por diante. Estas funções não necessitam estar definidas em um mesmo arquivo. A função `printf` foi invocada a partir de `main` no primeiro exemplo deste capítulo, mas sua definição não estava no mesmo arquivo. A fase de ligação do compilador é que é responsável em descobrir onde estão as definições das funções sendo usadas para poder criar um programa executável.

Este mecanismo de ligação de arquivos através de invocação de funções é que permite desenvolver aplicações complexas a partir de blocos mais simples. Idealmente, cada arquivo fonte deverá ter apenas algumas funções que estejam de alguma forma relacionadas (um *módulo*). Outros arquivos não precisam conhecer como aquelas funções estão internamente implementadas; para usá-las, basta conhecer como as funções utilizadas devem ser invocadas e que tipo de retorno é esperado da função (a *interface* da função).

2.5.3 A Função `printf`

A função `printf` já foi utilizada para imprimir um string na tela. Na verdade, esta função que é parte da biblioteca padrão de rotinas da linguagem C permite apresentar não apenas strings mas os valores de qualquer tipo de dado. Para tanto, `printf` utiliza o mecanismo de *formatação*, que permite traduzir a representação interna de variáveis para a representação ASCII que pode ser apresentada na tela.

O primeiro argumento de `printf` é um *string de controle*. Este string, que sempre deve estar presente, pode especificar através de caracteres especiais (as *sequências de conversão*) quantos outros argumentos estarão presentes nesta invocação da função. Estes outros argumentos serão variáveis cujos valores serão formatados e apresentados na tela.

Observe que `printf` não tem um número fixo de argumentos. A única forma de saber qual o número de variáveis que será apresentado é por inspeção do string de controle. Desta forma, cuidado deve ser tomado para que o número de variáveis após o string de controle esteja de acordo com o número de sequências de conversão presente no string de controle.

Além de ter o número correto de argumentos e sequências de conversão, o tipo de cada variável deve estar de acordo com a sequência de conversão especificada no string de controle. A sequência de conversão pode ser reconhecida dentro do string de controle por iniciar sempre com o caracter '%'. Por exemplo, supondo

que `x` seja uma variável do tipo `int`, seu valor poderia ser apresentado na tela por

```
printf ("Valor de x = %d\n", x);
```

Se o valor da variável `x` fosse 10 neste exemplo, então esta chamada a `printf` apresentaria na tela a frase

```
Valor de x = 10
```

seguido pelo caracter de nova linha (`\n`), ou seja, a próxima saída para a tela aconteceria na linha seguinte. Observe que a sequência de conversão pode ocorrer dentro de qualquer posição dentro do string de controle.

As principais sequências de conversão para variáveis caracteres e inteiras são:

`%c` imprime o conteúdo da variável com representação ASCII;

`%d` imprime o conteúdo da variável com representação decimal com sinal;

`%u` imprime o conteúdo da variável com representação decimal sem sinal;

`%o` imprime o conteúdo da variável com representação octal sem sinal;

`%x` imprime o conteúdo da variável com representação hexadecimal sem sinal.

Uma largura de campo pode ser opcionalmente especificada logo após o caracter `%`, como em `%12d` para especificar que o número decimal terá reservado um espaço de doze caracteres para sua representação. Se a largura de campo for negativa, então o número será apresentado alinhado à esquerda ao invés do comportamento padrão de alinhamento à direita. Para a conversão de variáveis do tipo `long`, o caracter `l` também deve ser especificado, como em `%ld`.

Para converter variáveis em ponto flutuante, as sequências são:

`%f` imprime o conteúdo da variável com representação com ponto decimal;

`%e` imprime o conteúdo da variável com representação em notação científica (exponencial);

`%g` formato geral, escolhe a representação mais curta entre `%f` e `%e`.

Como para a representação inteira, uma largura de campo pode ser especificada para números reais. Por exemplo, `%12.3f` especifica que a variável será apresentada em um campo de doze caracteres com uma precisão de três dígitos após o ponto decimal.

Finalmente, se a variável a ser apresentada é uma sequência de caracteres (um string), então o formato de conversão `%s` pode ser utilizado. Para apresentar o caracter `%`, a sequência `%%` é utilizada.

2.5.4 Escopo de Variáveis

Variáveis são normalmente declaradas no início do corpo de uma função. Tais variáveis são chamadas de *variáveis locais*, indicando que são de propriedade exclusiva da função. (Em C, estas variáveis são da classe de armazenamento automática.) Isto quer dizer que outras funções não podem ter acesso a estas variáveis. Por exemplo, no seguinte programa

```
main() {
    /* declara x e inicializa seu valor */
    int x=0;
    void xprint(); /* um prototipo */

    xprint();      /* chamada de funcao */
}

/* definicao da funcao que imprime valor de x */
void xprint() {
    printf("Valor de x = %d\n", x); /* erro! */
}
```

um erro de compilação seria acusado pois a função `xprint` não sabe o que é `x`, uma vez que esta variável é local à função `main`.

Há duas formas de contornar uma situação como esta. A primeira seria modificar a função `xprint` de forma que ela recebesse a variável a imprimir como um argumento, como em

```
main() {
    /* declara x e inicializa seu valor */
    int x=0;
    void xprint(int); /* um prototipo */

    xprint(x);      /* chamada de funcao */
}

/* definicao da funcao que imprime valor de x */
void xprint(int x) {
    printf("Valor de x = %d\n", x);
}
```

Neste caso, não há mais erro de compilação, uma vez que agora a variável `x` especificada em `printf` é uma variável conhecida por `xprint`. É importante observar, entretanto, que esta é uma *outra* variável `x`, agora de propriedade da função `xprint`. Para verificar este comportamento, o programa pode ser modificado uma vez mais:

```
main() {
    /* declara x e inicializa seu valor */
    int x=0;
    void xprint(int); /* um prototipo */

    xprint(x);          /* chamada de funcao */
    printf("Valor de x em main = %d\n", x);
}

/* definicao da funcao que imprime valor de x */
void xprint(int x) {
    x += 123;           /* incrementa o valor de x */
    printf("Valor de x em xprint = %d\n", x);
}
```

Este programa, se compilado e executado, deve produzir a seguinte saída:

```
Valor de x em xprint = 123
Valor de x em main = 0
```

Observe que a modificação realizada sobre `x` em `xprint` não se reflete na variável `x` declarada em `main`. Como já observado, argumentos em funções C são passados por valor (isto é, a variável `x` de `xprint` irá receber uma cópia do valor da variável `x` de `main`), e têm o mesmo comportamento de variáveis locais da função — só têm existência definida durante a execução da função.

Para entender porque uma variável local só pode ser referenciada durante o curso de existência de uma função, é preciso entender como estas variáveis são alocadas em memória. Um programa está associado a um processo do sistema operacional. Cada vez que uma função é ativada (invocada), ela “ganha” um espaço no segmento de pilha (Seção 1.3) onde os argumentos são passados. Além dos argumentos, este espaço na pilha também armazena as variáveis locais. Quando a função encerra sua execução, seu espaço é devolvido para o sistema operacional, e será reaproveitado pela próxima função chamada. Por este motivo, variáveis locais são ditas ser da classe de armazenamento automática, e não há condições de referenciar variáveis internas a funções após suas execuções.

A outra forma de contornar a situação original de erro é definir a variável `x` como uma variável compartilhada entre as duas funções, `main` e `xprint`. Neste caso, `x` passa a ser uma *variável global*:

```
int x;          /* declara x como global */

main() {
```

```

void xprint(); /* um prototipo */

xprint();      /* chamada de funcao */
}

/* definicao da funcao que imprime valor de x */
void xprint() {
    printf("Valor de x = %d\n", x);
}

```

Observe que `x` é declarada fora de qualquer função. Neste caso, qualquer função que for definida após a declaração de `x` irá “enxergar” esta variável — a não ser que a função defina uma outra variável local de nome `x`, quando então a variável local esconderia a definição global.

Variáveis globais são armazenadas no segmento de dados estáticos do processo, tendo assim uma existência independente da duração de cada função.

Variáveis Externas

Como já observado, um programa C pode estar dividido entre diversos módulos (ou diversos arquivos fonte). É possível compartilhar variáveis entre distintos módulos — para tanto, é preciso indicar que uma variável foi definida em outro módulo através do uso da palavra chave `extern`.

Considere o seguinte variação do exemplo anterior, onde as rotinas `main` e `xprint` são declaradas em dois arquivos distintos. O primeiro arquivo, `arq1.c`, contém a definição de `main` e da variável global `x`:

```

/*
 * arq1.c
 */
int x;      /* declara x como global */

main() {
    void xprint(); /* um prototipo */

    xprint();      /* chamada de funcao */
}

```

O segundo arquivo, `arq2.c`, define a função `xprint` que deverá acessar a variável global `x` que foi definida em `arq1.c`. Para tanto, `x` é declarada em `arq2.c` como `extern`:

```
/*
 *  arq2.c
 */
extern int x;

/* definicao da funcao que imprime valor de x */
void xprint() {
    printf("Valor de x = %d\n", x);
}
```

Ao ver `x` declarada como `extern` em `arq2.c`, o compilador sabe que esta variável estará sendo definida em outro módulo. Na fase de ligação, estas referências externas serão resolvidas para a criação do programa executável.

Variáveis Estáticas

C suporta um modificador de classe de armazenamento através do uso da palavra chave `static`. Variáveis do tipo `static` podem ser tanto internas a uma função quanto externas.

Variáveis locais estáticas (internas à função) passam a ter sua existência independente da ativação da função. Sob o ponto de vista dos segmentos de memória, são variáveis que, apesar de ainda ser propriedade exclusiva da função que as declaram, estão armazenadas no segmento de dados estáticos e não na pilha. Por exemplo,

```
int incr_cont() {
    static int contador=0;

    return(contador++);
}
```

Neste exemplo, `contador` é uma variável estática local à função `incr_cont`. Sendo uma variável estática, seu espaço em memória é fixo (presente no segmento de dados), de forma que é possível que a variável retenha seu valor entre distintas ativações da função (ao contrário do que ocorre com variáveis locais automáticas). Cada ativação da função `incr_cont` retorna o valor atual do `contador` e o incrementa, sendo este novo valor preservado até a próxima ativação da função.

Se uma variável global é declarada como estática, o efeito é de que esta variável só poderá ser vista dentro do módulo (arquivo fonte) onde ela foi definida. Em outras palavras, uma variável global estática não poderá ser vista como `extern` em outros módulos.

Uma função também pode ser definida como estática — o efeito é de que esta função só poderá ser invocada a partir de outras funções do mesmo módulo, tornando-se uma espécie de função local.

Como se pode observar, `static` em C não especifica simplesmente a classe de armazenamento, mas também um certo grau de privacidade. Variáveis estáticas locais só são conhecidas pelas funções que as declaram. Funções e variáveis globais estáticas só são conhecidas pelos módulos que as definem, sendo que seus nomes não irão interferir com outras variáveis e funções do mesmo nome em outros módulos.

2.5.5 Criando um Programa Executável

Como observado no Capítulo 1, para que o computador entenda um programa escrito em uma linguagem de alto nível é preciso traduzir as instruções deste programa para a linguagem do computador. A primeira etapa é a criação de um arquivo com o chamado *código fonte*. O código fonte contém as funções com instruções em linguagem C, que serão posteriormente compiladas em um arquivo com *código objeto*.

Para criar um arquivo com o código fonte do programa, é preciso usar um editor de programas ou editor de textos (desde que seja possível criar um arquivo sem caracteres de formatação). O Apêndice B apresenta um editor muito utilizado em ambiente Unix (já com porte para diversos outros sistemas), `emacs`. Este editor apresenta entre outras vantagens algumas facilidades para a edição de código C.

Após criado o arquivo com código fonte, este será passado como entrada para o programa compilador. O compilador C executa a tradução do código fonte para um arquivo executável em diversas etapas, quais sejam:

Pré-processador: nesta primeira etapa, algumas abreviaturas que o programador usou em seu código fonte são traduzidas para os comandos (não abreviados) da linguagem C. O resultado desta fase é ainda um código fonte, porém expandido. Estas abreviaturas são introduzidas no código fonte através das *diretivas do pré-processador*;

Compilador (propriamente dito): nesta segunda etapa, o código fonte é traduzido em linguagem assembly da máquina para a qual ele está sendo compilado. Nesta etapa verifica-se se o código fonte está sintaticamente correto, isto é, se as regras da linguagem de programação C foram seguidas. Otimizações sobre o código fonte são realizadas nesta fase;

Montador: usando este aplicativo do sistema, o código assembly (ainda um arquivo texto) é traduzido em linguagem de máquina (um arquivo binário).

O resultado é o *código objeto relocável*, um formato intermediário que não pode ser lido pelo usuário e não pode ser executado pelo sistema;

Ligador: este aplicativo do sistema executa a fase final da criação de um arquivo executável. Nesta etapa, o código objeto relocável do programa criado pelo usuário é ligado a rotinas internas do sistema (e possivelmente rotinas de outras aplicativos) para criar uma unidade independente, um arquivo com *código executável*.

Em um sistema operacional como Unix, estas etapas são transparentes para o usuário, pois o aplicativo compilador C (cc, descrito sucintamente no Apêndice C) realiza as quatro etapas internamente. Assim, o processo de criação de um programa C envolve a edição (criação do código fonte), a compilação (criação do código executável) e a execução do programa. Caso haja erros de sintaxe, o programa executável não será gerado, e deve-se voltar ao processo de edição para corrigir estes erros. Caso o programa executável tenha sido gerado mas apresente resultados imprevistos, o ciclo também deve se repetir a partir da edição.

2.6 Atividades

Atividade 1 Edite e compile o arquivo `ser.c`, e execute o programa `ser`.

Atividade 2 Descubra que tipo de mensagens o compilador irá apresentar para cada uma das seguintes situações:

- (a) um programa que só contém uma função cujo nome seja diferente de `main`;
- (b) uma expressão onde o terminador `;` tenha sido esquecido;
- (c) uma tentativa de se alterar uma variável declarada como `const`;
- (d) uma variável que é utilizada mas que não foi declarada;
- (e) a expressão `y = x;`, onde `x` e `y` são de um mesmo tipo, ocorre sem que o valor de `x` tenha sido definido;
- (f) a expressão `y = x;` ocorre, porém `x` é uma variável do tipo `float` enquanto que `y` é do tipo `char`;
- (g) uma variável é declarada com um nome inválido;
- (h) um bloco em um dos comandos de controle estruturado não foi apropriadamente encerrado (um `}` foi esquecido).

Atividade 3 A função `int rand()` do sistema retorna um número inteiro aleatório. Use esta rotina para desenvolver um programa que gere e imprima na tela uma sequência de 120 bits gerados aleatoriamente. Execute o programa diversas vezes.

Atividade 4 Uma função é *recursiva* quando ela invoca a si própria. Desenvolva uma função recursiva para calcular fatorial $n!$ de um número inteiro n , onde

$$n! = n \times (n - 1)!$$

Use esta função em um programa que imprima o fatorial dos primeiros 30 inteiros.

Atividade 5 Use a tabela ASCII do Apêndice A para criar funções que recebem um argumento `character` e:

- (a) retorne 1 um `character` é um dígito decimal, 0 em caso contrário;
- (b) retorne 1 se um `character` é uma letra, 0 se não;
- (c) retorne 1 se um `character` é uma letra ou dígito, 0 se não;
- (d) retorne 1 se um `character` é uma imprimível, 0 se não;
- (e) converta uma letra para maiúscula;
- (f) converta uma letra para minúscula;
- (g) retorne o valor inteiro correspondente a um dígito decimal.

Atividade 6 Desenvolva uma função que receba um arranjo de caracteres contendo uma sequência de dígitos decimais terminadas pelo `character '0'` e retorne o valor inteiro correspondente.

Atividade 7 Desenvolva uma função que receba um arranjo de caracteres contendo uma sequência de dígitos decimais e opcionalmente um ponto terminadas pelo `character '0'` e retorne o valor em ponto flutuante (`double`) correspondente.

Capítulo 3

Apontadores

Apontadores constituem um dos recursos mais utilizados na programação C. Eles fornecem um mecanismo poderoso, flexível e eficiente de acesso a variáveis. Há computações que só podem ser expressas através do uso de apontadores.

3.1 Variáveis Apontadores

Um apontador é uma variável que contém um endereço de outra variável. Este conceito é amplamente utilizado por diversos computadores, estando diretamente relacionado ao modo de endereçamento indireto (Seção 1.2.2).

Este mecanismo deve ser utilizado com critério e disciplina. O uso descuidado de apontadores pode levar a situações onde um endereço inválido é acessado, levando a erros de execução de programas.

Para definir que uma variável vai guardar um endereço, o operador unário `*` é utilizado na declaração, como em

```
/* define um apontador para um inteiro */
int *ap;
/* algumas variaveis inteiras */
int x, y;
```

Neste exemplo, `ap` é uma variável do tipo *apontador para inteiro*, ou seja, ela irá receber um endereço de uma variável inteira. Para se obter o endereço de uma variável, o operador unário `&` pode ser utilizado. No exemplo acima,

```
/* ap recebe o endereco de x */
ap = &x;
```

Após esta instrução, diz-se que `ap` *aponta para* `x`. É possível acessar o valor da variável `x` através do apontador usando o operador unário `*`, como em

```
/* y recebe o conteudo da variavel apontada por ap */  
y = *ap;
```

Observe que a combinação `*ap` é um inteiro, que pode assim ser atribuído a outro inteiro. Esta associação pode facilitar a compreensão da declaração de apontadores. No exemplo, `int *ap` pode ser lido como “`*ap` é um inteiro.”

Também seria possível definir o valor de `x` através do apontador, como em

```
/* o conteudo da variavel apontada por ap recebe y */  
*ap = y;
```

Apontadores podem tomar parte em expressões aritméticas. Assim, a seguinte expressão é perfeitamente válida:

```
y = *ap + 1;
```

ou seja, `y` receberia o valor de `x` (variável apontada por `ap`) incrementado de 1.

3.2 Aritmética de Apontadores

Não apenas o conteúdo de apontadores podem tomar parte em expressões aritméticas. C também suporta o conceito de operações sobre endereços, embora as operações que possam ser utilizadas neste caso sejam limitadas. Tais operações definem a *aritmética de apontadores*.

Para apresentar o conceito de aritmética de apontadores, considere o seguinte exemplo:

```
main() {  
    int arr[10];      /* arr: arranjo com 10 inteiros */  
    int *el;          /* el: apontador para um inteiro */  
    int i;  
  
    el = &arr[0];     /* inicializa apontador */  
  
    /* inicializa conteudo do arranjo via apontador */  
    for (i=0; i<10; ++i)  
        *(el + i) = 0;  
}
```

O apontador `el` aponta inicialmente para o primeiro elemento do arranjo `arr`, ou seja, `arr[0]` — `&arr[0]` é o endereço deste elemento. Assim, para acessar este elemento através do apontador, a expressão `*el` poderia ser utilizada. No entanto, é possível também acessar outros elementos do arranjo através do apontador.

Para acessar o elemento seguinte, a expressão `*(e1+1)` retorna o endereço do próximo inteiro armazenado após o endereço `e1`, ou seja, o endereço de `arr[1]`. Portanto, o que a instrução interna ao laço no exemplo está realizando é o acesso a cada elemento do arranjo através de um apontador.

Um aspecto fundamental da aritmética de apontadores é que ela libera o programador de saber qual a dimensão alocada para cada tipo de dado. Quando um apontador é incrementado, este incremento irá refletir o tamanho do tipo da variável que está sendo apontada. Assim, o exemplo acima funcionará independentemente da máquina no qual ele for executado, ocupe a representação de um inteiro dois ou quatro bytes. Quando a expressão `e1+i` é encontrada, o endereço do *i*-ésimo inteiro após o endereço `e1` é obtido — ou seja, esta expressão aponta para o elemento `arr[i]`.

A aritmética de apontadores está limitada a quatro operadores: soma, subtração, incremento e decremento. Outra limitação é que, no caso de soma, o outro operando além do apontador deve ser uma expressão (ou variável ou constante) inteira. A subtração de dois apontadores para um mesmo tipo de dado é uma operação válida, retornando o número de elementos entre os dois apontadores. A comparação entre dois apontadores também é uma operação legal.

Há uma única exceção ao uso legal de aritmética de apontadores: quando um apontador é definido para o tipo `void`. Neste caso, a variável apontador contém um endereço genérico, sobre um tipo que não pode ser determinado. Portanto, operações aritméticas sobre este tipo de apontador não são permitidas.

3.3 Apontadores e Arranjos

Como observado no exemplo acima, apontadores e arranjos estão intimamente relacionados em C. Na verdade, qualquer referência a um arranjo é convertida internamente para uma referência do tipo apontador. Por este motivo, quando eficiência de tempo de acesso é uma preocupação, muitos programadores trabalham diretamente com apontadores.

O nome de um arranjo é uma expressão do tipo apontador que corresponde ao endereço do primeiro elemento do arranjo. Assim, a inicialização do apontador no exemplo acima poderia ser reescrita como

```
e1 = arr;    /* arr equivale a &arr[0] */
```

Observe que, uma vez que `arr` equivale a um apontador, o elemento `arr[i]` poderia ser acessado da mesma forma como `*(arr+i)`. Na verdade, é isto que o compilador irá fazer internamente: qualquer expressão da forma `E1[E2]` será internamente traduzida para `*((E1)+(E2))`. Observe que isto implica que esta

operação de indexação é comutativa, embora tal fato raramente seja utilizado em programação C.

Por outro lado, o inverso (usar o operador de indexação com uma variável apontador) também é possível. Assim, o laço de atribuição no exemplo acima poderia ter sido escrito como

```
for (i=0; i<10; ++i)
    el[i] = 0;
```

Apesar da forma usando o operador `*` ser mais eficiente, programadores iniciantes muitas vezes acham mais simples entender o acesso usando o operador de indexação, e acabam preferindo esta forma.

Uma diferença fundamental entre um apontador e o nome de um arranjo é que o apontador é uma *variável*, enquanto que o nome de um arranjo é uma *constante*. Assim, expressões como `arr++` ou `&arr` não fazem sentido.

Outra diferença que deve ser ressaltada é o fato de que a declaração de um arranjo efetivamente reserva o espaço para as variáveis, enquanto que a declaração de um apontador reserva apenas espaço para guardar um endereço. Considere o seguinte exemplo:

```
/*
 * Exemplo do uso indevido de apontador
 */
main() {
    int *el;                /* el: apontador para inteiro */
    int i;

    /* inicializa conteudo */
    for (i=0; i<10; ++i)
        el[i] = 0;         /* onde esta el[i]? */
}
```

Uma vez que o apontador `el` não foi inicializado, a expressão `el[i]` pode estar apontando para qualquer posição da área de memória — possivelmente, para alguma posição inválida. Observe que o fato de ter declarado o apontador não significa que esta variável possa ser utilizada como um arranjo. Para tal, o apontador deve estar com o endereço de alguma posição válida, seja através de uma atribuição envolvendo um arranjo, seja através do uso de rotinas de alocação dinâmica (Seção 3.7).

Uma vez que apontadores são variáveis, nada impede que arranjos de apontadores sejam definidos. De fato, uma declaração tal como

```
int *aa[10];
```

define uma variável `aa` que é um arranjo de dez apontadores para variáveis inteiras. Cada elemento deste arranjo, desde `aa[0]` até `aa[9]`, é um apontador para inteiro(s) que tem a mesma propriedade que os apontadores vistos até o momento.

Observe que esta forma suporta uma alternativa para trabalhar com arranjos multidimensionais, desde que respeitadas as diferenças entre apontadores e arranjos. Arranjos de apontadores trazem uma flexibilidade adicional pelo fato de que cada “linha” pode ter tamanho variável. No exemplo acima, cada apontador `aa[i]` pode estar apontando para um inteiro, para o primeiro elemento de um arranjo com diversos inteiros, ou mesmo para nenhum inteiro.

3.4 Argumentos na Linha de Comando

Até o momento, a definição da função `main` foi sempre utilizada em sua forma sem parâmetros. Há, no entanto, uma outra forma de definição de `main` que é utilizada para passar para o programa quais argumentos foram dados na linha de comando do sistema operacional. Nesta outra forma, `main` recebe dois parâmetros, como indicado abaixo:

```
int main(int argc, char *argv[]) {  
    ...  
}
```

O primeiro parâmetro, `argc`, indica o número de *tokens* presente na linha de comando. Por exemplo, uma chamada a um programa de nome `eco` com dois argumentos, como

```
$ eco um dois
```

faria com que o valor de `argc` passado para a função `main` fosse igual a três.

O segundo parâmetro, `argv`, é um arranjo de strings, onde cada elemento do arranjo representa um dos tokens da linha de comando. Assim, no exemplo acima a função `main` receberia os seguintes strings nesta variável:

- `argv[0]`: o string `"eco"`;
- `argv[1]`: o string `"um"`;
- `argv[2]`: o string `"dois"`.

Observe que `argv[0]` sempre armazenará o nome do programa sendo executado, enquanto que `argv[i]` armazenará o *i*-ésimo argumento passado para o programa, para *i* variando desde 1 até `argc-1`.

Um exemplo simplificado de uma rotina que ecoe seus argumentos é apresentado abaixo, ilustrando o uso de `argc` e `argv`:

```

/*
 * eco.c: repete os argumentos da linha de comando
 */
main(int argc, char *argv[]) {
    int i = 1;

    /* apresenta todos argumentos a partir de argv[1] */
    while (i < argc) {
        printf("%s ", argv[i]);
        ++i;
    }
    /* terminando, passa para proxima linha */
    printf("\n");
}

```

3.5 Apontador como Argumento de Funções

A passagem por valor, padrão em C, não permite que uma função manipule diretamente uma variável que lhe esteja sendo passada. Quando se deseja manipular diretamente a variável, apontadores devem ser usados como o recurso de acesso.

Para ilustrar esta condição, imagine como implementar a rotina `swap` que recebe dois argumentos de um mesmo tipo e troca seus valores. (Esta função poderia fazer parte de uma rotina de ordenação de elementos.) Por exemplo, para trocar dois inteiros, algo similar à seguinte função seria desejado:

```

/*
 * funcao (errada) de troca de inteiros
 */
void swap_err(int e11, int e12) {
    int temp;          /* variavel temporaria */

    temp = e11;
    e11 = e12;
    e12 = temp;
}

```

Entretanto, como observado no comentário inicial, esta função não funciona. Supondo que a função `main` de um programa tente acessar esta rotina, como em

```
main() {
```

```

    int a=10,
        b=20;
    void swap_err(int, int);

    printf("a=%d, b=%d\n", a, b);
    swap_err(a, b);
    printf("a=%d, b=%d\n", a, b);
}

```

A saída obtida seria:

```

a= 10, b= 20
a= 10, b= 20

```

Como se observa, a função `swap_err` não realiza a troca de valores das variáveis `a` e `b` de `main`, apesar de sua lógica interna estar correta. O que `swap_err` faz é trocar os valores das *cópias* destas variáveis, que são apenas suas variáveis locais.

A fim de se obter o efeito correto, apontadores devem ser utilizados como argumentos. Assim, os elementos a serem trocados serão acessados por seus endereços, e seus conteúdos serão efetivamente alterados. Nesta nova versão, a função `swap` é definida como:

```

/*
 * funcao de troca de inteiros
 */
void swap(int *el1, int *el2) {
    int temp;          /* variavel temporaria */

    temp = *el1;
    *el1 = *el2;
    *el2 = temp;
}

```

A chamada à função deve passar os endereços das variáveis, como em

```

main() {
    int a=10,
        b=20;
    void swap(int *, int *);

    printf("a=%d, b=%d\n", a, b);
    swap(&a, &b);
    printf("a=%d, b=%d\n", a, b);
}

```

A saída obtida neste caso seria:

```
a= 10, b= 20
a= 20, b= 10
```

como desejado.

Outros usos de apontadores como argumentos incluem funções que devem retornar mais de um valor e a passagem de arranjos para funções. Quando um arranjo é passado para uma função, na verdade o que se passa é o endereço de seu primeiro elemento. Por este motivo, é possível omitir qual a dimensão do arranjo na declaração do tipo do argumento, como foi visto no caso de `argv` (Seção 3.4). Quando o argumento é um arranjo multidimensional, apenas a dimensão do primeiro índice pode ser omitida — as demais devem ser fornecidas. Caso contrário, seria impossível saber como acessar corretamente os elementos do arranjo.

3.6 Apontadores para Funções

Como foi visto no Capítulo 1, um programa é um conjunto de instruções armazenado na memória, assim como seus dados. Por este motivo, é possível referenciar o *endereço de uma função*. Em C, o endereço de uma função é acessível ao programador através de uma variável do tipo apontador para função.

Apontadores para funções podem ser passados como argumentos para outras funções, e a função apontada pode ser invocada a partir de seu apontador. Um exemplo prático desta capacidade é seu uso em uma rotina de ordenação de elementos de um arranjo. Se o arranjo é de inteiros, então uma função de comparação de inteiros deverá ser suportada, tal como

```
/*
 * compara dois inteiros, retornando:
 *      0 se os dois elementos forem iguais
 *      um inteiro negativo se o primeiro elemento for menor
 *      um inteiro positivo se o primeiro elemento for maior
 */
int comp_int(int *e1, int *e2) {
    return(*e1 - *e2);
}
```

O problema surge quando se deseja usar o mesmo algoritmo de ordenação para ordenar outros arranjos de tipos que não sejam inteiros. Por exemplo, se os elementos a comparar forem strings, então a rotina de comparação acima não mais serviria, apesar de todo o restante do algoritmo de ordenação ainda ser basicamente o mesmo.

A solução é passar qual função deve ser usada para a comparação como um dos argumentos para a rotina de ordenação genérica. Esta abordagem é adotada por rotinas usualmente supridas juntamente com o compilador C, tal como `qsort` para ordenação de arranjos e `bsearch` para a realização de busca binária em arranjos ordenados.

A forma de declarar uma variável do tipo apontador para função é ilustrada no seguinte exemplo, com uma referência à função `comp_int` definida acima:

```
main() {
    /* prototipo de comp_int: */
    int comp_int(int *, int *);
    /* apontador para uma funcao retornando inteiro */
    int (*apcmp)();
    int a, b;

    apcmp = comp_int;          /* inicializa apontador */
    ...
    (*apcmp)(a, b);            /* invoca funcao */
}
```

Algumas observações relativas a este exemplo são importantes. A primeira refere-se à declaração do apontador. A declaração de um apontador para a função deve incluir os parênteses em torno do nome da variável apontador. Uma definição na forma `int *apcmp()`; seria interpretada como o protótipo de uma função retornando um apontador para um inteiro, o que não é o desejado neste caso.

A segunda observação refere-se à forma utilizada para definir o valor do apontador no comando de atribuição. Como o protótipo de `comp_int` já havia sido definido, então o compilador sabe que este identificador refere-se a uma função. Quando o identificador `comp_int` é encontrado novamente, desta vez sem parênteses, ele é identificado como o endereço desta função, podendo assim ser atribuído a um apontador para uma função com o mesmo tipo de retorno. Repare a semelhança com referências a nomes de arranjos.

Finalmente, a invocação da função através de seu apontador: a forma usando o operador de dereferência `(*apcmp)` indica o conteúdo do apontador `apcmp`, que é a função (neste caso, `comp_int`). Assim, a última linha no exemplo é apenas uma invocação para a rotina apontada por `apcmp`, e o que vem a seguir de `(*apcmp)` são simplesmente os argumentos para a função. O padrão ANSI também permite que a forma equivalente,

```
apcmp(a, b);
```

seja utilizada. Muitos programadores preferem a forma apresentada no exemplo original para tornar claro que um apontador para função está sendo usado, embora

internamente não haja diferenças entre a ativação de uma função por seu nome ou através de um apontador.

Apontadores para funções tornam-se interessantes quando o programador não pode determinar qual função deve ser executada em uma dada situação a não ser durante a execução do programa. Em tais casos, o trecho do programa referenciando esta “função variável” pode ser escrito em termos de ativação de uma função através de apontadores para funções, os quais são corretamente inicializados em tempo de execução.

3.7 Gerenciamento da Memória Livre

Um exemplo marcante do uso de apontadores em C está no gerenciamento da área livre no segmento de dados de um processo. Como já observado, o segmento de dados tem uma área que pode ter sua dimensão dinamicamente modificada segundo as necessidades do processo. A linguagem C permite que o programador utilize este recurso do sistema operacional através de suas rotinas de *alocação dinâmica de memória*, que estão presentes na biblioteca padrão da linguagem.

Há duas atividades básicas relacionadas à manipulação desta área:

1. o programa pode requisitar uma área de memória dentro do espaço livre disponível; ou
2. o programa pode liberar uma área de memória que tenha sido previamente requisitada do espaço livre e que não seja mais necessária.

Em C, a alocação de memória é suportada pela rotina `malloc`. Esta rotina recebe um argumento, que é a dimensão em bytes da área necessária. O valor de retorno é o endereço do início da área que o sistema operacional alocou para este pedido.

Para um exemplo do uso de `malloc`, suponha que `n` é uma variável que descreve o número de inteiros que serão necessários para armazenar dados de uma dada tarefa. O programador não teve como determinar o valor de `n`, de forma que este valor é obtido em tempo de execução. Para reservar o espaço necessário, alocação dinâmica é utilizada:

```
main() {
    unsigned int n;      /* tamanho em inteiros da area */
    int *area;           /* apontador para inicio da area */
    void *malloc();      /* prototipo de malloc */

    /* obtem valor de n */
    ...
```

```
/* obtem espaco da area livre */
area = malloc(n*sizeof(int));

/* trabalha com a area alocada */
...
}
```

Observe que o protótipo de `malloc` indica que esta função retorna um apontador para o tipo `void`. Uma variável apontador para `void` denota um endereço genérico, que pode ser atribuído a um apontador para qualquer outro tipo sem problemas de conversão.

O conteúdo inicial da área alocada é indefinido. Uma variante da rotina de alocação, chamada `calloc`, inicializa o conteúdo da área alocada para 0's. Esta rotina recebe dois argumentos ao invés de um: o primeiro argumento é o número de itens que será alocado, e o segundo é o tamanho em bytes de cada item. Usando `calloc`, o exemplo acima seria reescrito como

```
main() {
    unsigned int n;      /* tamanho em inteiros da area */
    int *area;           /* apontador para inicio da area */
    void *calloc();       /* prototipo de calloc */

    /* obtem valor de n */
    ...

    /* obtem espaco da area livre */
    area = calloc(n, sizeof(int));

    /* trabalha com a area alocada */
    ...
}
```

Para liberar uma área alocada por `malloc` ou `calloc`, a rotina `free` deve ser utilizada. Assim, o exemplo acima poderia ser complementado da seguinte forma:

```
main() {
    unsigned int n;      /* tamanho em inteiros da area */
    int *area;           /* apontador para inicio da area */
    void *calloc();       /* prototipo de calloc */
    void free();          /* prototipo de free */
}
```

```

/* obtem valor de n */
...

/* obtem espaco da area livre */
area = calloc(n, sizeof(int));

/* trabalha com a area alocada */
...

/* libera a area pre-alocada */
free(area);
}

```

O que a rotina `free` faz é retornar o espaço que havia sido pré-alocado dinamicamente para o gerenciador de espaço livre, que pode assim reutilizar esta área para atender a outras requisições.

Há ainda uma quarta rotina associada ao gerenciamento de espaço livre em C, `realloc`, que permite alterar a dimensão de uma área pré-alocada. Esta rotina recebe dois argumentos, o primeiro sendo o apontador para a área que já havia sido alocada e o segundo sendo a nova dimensão para esta área em bytes. O valor de retorno é o endereço (que pode eventualmente ser o mesmo que o original) para a área com a nova dimensão requisitada. Caso o endereço seja diferente, `realloc` se encarrega de copiar o conteúdo original para a nova área alocada. Por exemplo, para dobrar o tamanho original apontada por `area` no código acima,

```

main() {
    unsigned int n;      /* tamanho em inteiros da area */
    int *area;           /* apontador para inicio da area */
    void *calloc();      /* prototipo de calloc */
    void free();         /* prototipo de free */
    void *realloc();     /* prototipo de realloc */

    /* obtem valor de n */
    ...

    /* obtem espaco da area livre */
    area = calloc(n, sizeof(int));

    /* trabalha com a area alocada */
    ...

    /* dobra a area original */

```

```
    area = realloc(area, 2*n*sizeof(int));

    /* continua trabalho com a area alocada */
    ...

    /* libera a area pre-alocada */
    free(area);
}
```

3.8 O Apontador Nulo

Nos exemplos da seção anterior, não se comentou o que aconteceria caso não houvesse espaço disponível em memória para atender à requisição de alocação dinâmica. Para lidar com tais situações, C introduz o conceito de *apontador nulo*.

Um apontador nulo é um valor especial que não corresponde a nenhum endereço válido para variáveis. Quando `malloc` (`calloc`, `realloc`) não consegue obter o espaço requisitado, o valor retornado pela função é o apontador nulo. Este é um comportamento típico em C para indicar erros em rotinas que retornam apontadores.

Internamente, o compilador mantém um apontador nulo para cada tipo de dado. Felizmente, o programador não precisa saber como representar estes valores internos — ele usa simplesmente a constante 0, e esta constante é internamente convertida para o apontador nulo durante a compilação. Assim, o exemplo anterior pode ser estendido para verificar situações de erro como se segue:

```
main() {
    unsigned int n; /* tamanho em inteiros da area */
    int *area;      /* apontador para inicio da area */
    void *calloc(); /* prototipo de calloc */
    void free();    /* prototipo de free */
    void *realloc(); /* prototipo de realloc */

    /* obtem valor de n */
    ...

    /* obtem espaco da area livre */
    area = calloc(n, sizeof(int));
    if (area == 0) {
        printf("Erro de alocao\n");
        return(1);
    }
}
```

```
    }

    /* trabalha com a area alocada */
    ...

    /* dobra a area original */
    area = realloc(area, 2*n*sizeof(int));
    if (area == 0) {
        printf("Erro de expansao\n");
        return(2);
    }

    /* continua trabalho com a area alocada */
    ...

    /* libera a area pre-alocada */
    free(area);

    /* tudo acabou bem... */
    return(0);
}
```

Este exemplo também ilustra uma outra convenção usual em programas C: a definição do valor de retorno de `main`. Em geral, um valor de retorno diferente de 0 servirá para indicar ao sistema operacional (ou a um outro processo que tenha ativado este programa) que alguma condição de erro ocorreu que impediu o programa de completar com sucesso sua execução; o valor de retorno 0 indica um a terminação com sucesso.

3.9 Atividades

Atividade 1 Desenvolva um programa `soma` que receba um número arbitrário de inteiros na linha de comando e imprima a soma destes números.

Atividade 2 Desenvolva um programa `opera` que receba como argumentos um símbolo indicando uma operação (+, -, *, /) e dois números (inteiros ou reais) retornando o resultado da operação.

Atividade 3 Estenda o programa `eco` de forma que ele aceite como argumentos variáveis de ambiente do sistema operacional precedidas pelo símbolo \$. Por exemplo,

```
>eco $path  
/usr/bin
```

Use a rotina do sistema `char *getenv(char *)`, que recebe como argumento o nome de uma variável do ambiente e retorna o string com o valor atual da variável.

Atividade 4 Desenvolva uma função que receba dois strings como argumento e retorne 1 se seus conteúdos forem iguais e 0 se forem diferentes.

Atividade 5 Desenvolva um programa `gerarand` que receba zero, um ou dois argumentos inteiros (n_1 e n_2) e apresente na saída uma sequência de n_1 números aleatórios na faixa de 0 a $n_2 - 1$. Se apenas um argumento estiver presente, não há restrições na faixa de valores dos números gerados (o valor retornado por `rand()` pode ser diretamente utilizado). Se nenhum argumento estiver presente, 20 números aleatórios sem restrição de faixa de valores são gerados e apresentados.

Atividade 6 Modifique o programa acima para que os números sejam apresentados em ordem. Utilize a rotina `qsort` do sistema para ordenar os números em ordem crescente.

Capítulo 4

Tipos Derivados

Além dos tipos básicos já vistos, C permite trabalhar com outros tipos derivados, tais como estruturas, uniões e enumerações. O mecanismo de definição de nomes de tipos permite ainda que tantos tipos básicos quanto derivados possam ser tratados de maneira uniforme.

4.1 Estruturas

Uma *estrutura* é uma coleção de uma ou mais variáveis tratadas sob um único nome. Conceitualmente, este conceito é similar ao conceito de registros, presente em diversas linguagens de programação (tal como *record* em Pascal).

4.1.1 Definição de Estruturas

A forma geral de definição de uma estrutura C é

```
struct tag {  
    /* declaracao de componentes: */  
    ...  
} var1, var2, ..., varN;
```

A palavra chave **struct** inicia a definição da estrutura. A etiqueta (**tag**) é opcional, porém deve estar presente caso se deseje referenciar esta estrutura em algum momento posterior. Da mesma forma, a lista de variáveis declaradas (**var1**, ..., **varN**) também não precisa estar presente — a não ser que a etiqueta não esteja presente. Quando a etiqueta está presente, variáveis com esta mesma estrutura podem ser definidas posteriormente, como em

```
struct tag varM;
```

que declara `varM` como sendo uma variável do tipo `struct tag`.

Considere a definição de uma entidade `data`. Uma `data` é composta por três componentes — dia, mês e ano — sendo cada componente um número inteiro. Em C, este conceito é representável como uma estrutura, que pode ser definida como:

```
struct data {
    int dia;
    int mes;
    int ano;
};
```

Variáveis deste tipo de estrutura podem ser definidas como

```
struct data hoje;
```

Apesar deste exemplo só ter componentes inteiros, qualquer tipo válido pode estar presente em uma estrutura — até mesmo outra estrutura, como em

```
struct dados_pessoais {
    char nome[40];
    struct data nascimento;
};
```

Da mesma forma que tipos básicos, estruturas podem ser passadas como argumentos ou ser valores de retorno de funções. Por exemplo, seria possível ter uma função que retornasse a idade em anos de uma pessoa recebendo como argumentos variáveis do tipo estrutura `dados_pessoais` (que contém a data de nascimento no exemplo acima) e do tipo estrutura `data`, como em

```
int main() {
    struct data hoje;
    struct dados_pessoais aluno;
    int idade;
    int calc_idade(struct dados_pessoais, struct data);

    /* obtem dados para hoje e aluno */
    ...
    idade = calc_idade(aluno, hoje);

    /* apresenta resultado */
    printf("Idade: %d\n", idade);

    return(0);
}
```

Observe que variáveis do tipo estrutura são tratadas exatamente da mesma forma que variáveis de tipos básicos. Assim, não deve ser surpreendente que arranjos de estruturas e variáveis do tipo apontador para estruturas possam ser definidos, como em

```
struct dados_pessoais alunos[100];
struct dados_pessoais *pa;
```

Em algumas situações, pode haver a necessidade de ter como um dos componentes da estrutura um apontador para um tipo da própria estrutura. Um exemplo típico é a construção de listas ligadas, compostas por nós onde um dos dados armazenados em cada nó é um apontador para o próximo nó. Esta situação seria representada como

```
struct no_lista {
    /* conteudo do no: */
    ...
    /* apontador ao proximo no: */
    struct no_lista *proximo;
};
```

4.1.2 Acesso a Elementos

Uma vez que uma estrutura tem componentes internos que devem ser acessados para processamento, algum mecanismo de acesso deve ser fornecido pela linguagem. C oferece dois operadores que permitem acessar elementos de estruturas.

O operador básico de acesso a elementos de estruturas é o operador `.` (ponto). Por exemplo, uma versão simplificada da função `calc_idade` (citada no exemplo anterior) poderia calcular a idade simplesmente como a diferença dos elementos `ano` das estruturas `hoje` e `nascimento`. Esta versão simples seria:

```
int calc_idade(struct dados_pessoais pessoa,
               struct data hoje) {
    return(hoje.ano - pessoa.nascimento.ano);
}
```

Observe que a componente `pessoa.nascimento` da estrutura `pessoa` é também uma estrutura, e para chegar ao componente desejado (`ano`) o operador ponto deve ser aplicado recursivamente. Uma vez que não há limites no número de níveis de aninhamento de estruturas, também não há limites no número de vezes que o operador ponto pode ser recursivamente aplicado. Este operador é associativo da esquerda para a direita.

A outra forma de acesso a membros de estruturas facilita a notação quando apontadores para estruturas estão envolvidos. Para ilustrar esta forma, suponha que uma função que lê os dados de um aluno no programa acima retorna na verdade um apontador para uma estrutura do tipo `dados_pessoais`. Por exemplo,

```
struct dados_pessoais *le_aluno();    /* prototipo */
struct dados_pessoais *aluno_pt;      /* apontador */

aluno_pt = le_aluno();
```

O acesso a membros da variável `aluno_pt` poderia ser feito da forma usual, ou seja, `*aluno_pt` é uma estrutura, então seria possível acessar seus membros como em

```
printf("%s nasceu em %2d/%2d/%4d\n",
      (*aluno_pt).nome,
      (*aluno_pt).nascimento.dia,
      (*aluno_pt).nascimento.mes,
      (*aluno_pt).nascimento.ano);
```

A notação simplificada utiliza o apontador `->` (seta) para substituir uma construção na forma `(*A).M` por `A->M`. O exemplo original poderia ser reapresentado integrando estes últimos aspectos como

```
int main() {
    struct data hoje;
    struct dados_pessoais *aluno_pt;
    int idade;
    /* prototipos: */
    int calc_idade(struct dados_pessoais, struct data);
    struct data le_hoje();
    struct dados_pessoais *le_aluno();

    /* obtém dados para hoje e aluno */
    hoje = le_hoje();
    aluno_pt = le_aluno();

    idade = calc_idade(*aluno_pt, hoje);

    /* apresenta resultado */
    printf("Idade de %s: %d\n",
          aluno_pt->nome,
          idade);
```

```
    return(0);  
}
```

As mesmas regras de acesso são válidas quando arranjos de estruturas são utilizados, tal como em

```
struct dados_pessoais alunos[100];  
int i;  
...  
for (i=0; i<100; ++i)  
    printf("[%3d] %s\n",i,alunos[i].nome);
```

4.1.3 Campos

Em situações onde há a necessidade de se minimizar o espaço ocupado por dados de um programa, pode ser preciso compartilhar uma palavra da máquina para armazenar diversas informações. Neste caso, é preciso trabalhar a nível de subsequências de bits internas à representação de um número inteiro.

A forma básica de manipular bits em um inteiro seria através dos operadores bit-a-bit, trabalhando com máscaras e deslocamentos para isolar as informações individuais. Entretanto, a linguagem C suporta o conceito de *campos* de bits internos a um inteiro para facilitar este tipo de manipulação. Campos são baseados em estruturas.

Por exemplo, considere uma instrução de um dado dispositivo que está sendo programado em C. Cada instrução neste dispositivo têm um formato de quatro bits que especificam a operação, seguidos por dois campos de seis bits cada que especificam a fonte e o destino associados à operação dada. Com o uso de campos, uma variável poderia ser definida como

```
struct {  
    unsigned int opcode: 4;  
    unsigned int fonte: 6;  
    unsigned int dest: 6;  
} instrucao;
```

Campos são acessados exatamente da mesma forma que membros de estruturas, comportando-se como inteiros sem sinal. Entretanto, campos não têm endereços, ou seja, uma expressão como `&instrucao.dest` não faz sentido.

4.2 Uniões

Uma *união* permite que uma dada área de memória seja tratada como variáveis de tipos diferentes em instantes de tempos diferentes. Uniões são definidas e acessadas de forma similar a estruturas, usando a palavra chave `union` ao invés de `struct`.

A título de exemplo, considere a situação onde se queira obter a representação interna do número em ponto flutuante 0.25. Um possível programa para imprimir esta representação em octal é

```
int main() {
    union {
        float f;
        unsigned int i;
    } num;

    num.f = 0.25;
    printf("%f tem representacao %o\n",
           num.f, num.i);
}
```

O compilador se encarrega de reservar espaço para armazenar dados de tamanho do maior dos membros da união.

4.3 Enumerações

Uma outra forma de tipo composto em C é a *enumeração*. Usualmente, faz parte do processo de desenvolvimento de um programa associar códigos numéricos a variáveis que podem assumir um único valor dentre um conjunto finito de opções. O tipo enumeração permite associar nomes descritivos a tais conjuntos de valores numéricos.

Considere uma extensão da estrutura `dados_pessoais` apresentada acima que incorporasse também o sexo da pessoa. Há dois estados possíveis para uma variável deste tipo: ela pode assumir o valor `masculino` ou o valor `feminino`. Uma enumeração que poderia representar este tipo de informação seria

```
enum sex { masculino, feminino };
```

Uma variável deste tipo de enumeração poderia ser então incorporada na estrutura `dados_pessoais`,

```
struct dados_pessoais {  
    char nome[40];  
    struct data nascimento;  
    enum sex genero;  
};
```

O seguinte trecho de programa ilustra como os nomes descritivos definidos em enumerações são utilizados como valores:

```
int calc_idade(struct dados_pessoais pessoa,  
              struct data hoje) {  
    int idade;  
  
    idade = hoje.ano - pessoa.nascimento.ano;  
    if (pessoa.genero == feminino)  
        idade -= 10;  
  
    return(idade);  
}
```

Internamente, o compilador designa o valor 0 para o primeiro símbolo da enumeração, e incrementa de um o valor associado a cada símbolo na sequência. Isto pode ser modificado se o programador quiser através de atribuição explícita de um valor inteiro a um símbolo, como em

```
enum cedula {  
    beijaflor = 1,  
    garca = 5,  
    arara = 10 };
```

4.4 Definição de Nomes de Tipos

Embora C não permita a criação de novos tipos de dados, ela oferece uma facilidade para criar novos nomes para os tipos existentes, sejam eles básicos ou derivados. Este mecanismo, `typedef`, permite principalmente melhorar a facilidade de compreensão de programas.

A forma geral de uma definição de nome de tipo é

```
typedef tipo novo_nome;
```

Por exemplo, os tipos de estruturas `data` e `dados_pessoais` definidos anteriormente poderiam ser associados a nomes de tipos `Data` e `Pessoa` respectivamente pelas declarações

```
typedef struct data Data;
typedef struct dados_pessoais Pessoa;
```

Com estas definições, as declarações do programa que apresenta a idade de pessoas poderiam ser reescritas como

```
/*
 * Exemplo calcula idade com definicao de nomes de tipos
 */
/* Define estruturas e nomes de tipos */
typedef enum sex {masculino, feminino} Sexo;

typedef struct data {
    int dia;
    int mes;
    int ano;
} Data;

typedef struct dados_pessoais {
    char nome[40];
    Data nascimento;
    Sexo genero;
} Pessoa;

int main() {
    Data hoje;
    Pessoa *aluno_pt;
    int idade;
    /* prototipos: */
    int calc_idade(Pessoa, Data);
    Data le_hoje();
    Pessoa *le_aluno();
    ...
}
```

Outros exemplos de uso de typedef são

```
typedef unsigned int Tamanho;
typedef enum {false=0, true} Boolean;
```

4.5 Atividades

- Atividade 1** Estenda a função `calcula_idade` de forma a melhorar o cálculo da idade, retornando uma `Data` dizendo qual a idade da pessoa em anos, meses e dias.
- Atividade 2** Desenvolva um program que receba três strings como argumento, sendo o primeiro um nome, o segundo uma data e o terceiro um caracter M ou F. Estes argumentos são usados para inicializar os dados do exemplo da Seção 4.4 e imprimir a idade da pessoa no formato da atividade anterior.
- Atividade 3** Estenda o programa `gerarand` do capítulo anterior de forma que a saída, além de ordenada, apresente após cada número aleatório um número entre colchetes que indique a ordem em que o número foi gerado. Assim, o primeiro número aleatório gerado estaria associado à [1], o segundo a [2] e assim por diante,

Capítulo 5

O Pré-processador C

Há uma série de definições que podem coexistir em diversos módulos de um mesmo programa: valores constantes que devem ser compartilhados, definição de estruturas, definições de nomes de tipos, protótipos de funções, etc. Seria tedioso e muito sujeito a erros se estas definições tivessem de ser inseridas em cada módulo.

A linguagem C oferece mecanismos que permitem manter definições unificadas que são compartilhadas entre diversos arquivos. A base destes mecanismos é o pré-processamento de código, a primeira fase na compilação do programa. Nesta fase, por exemplo, comentários são substituídos por espaços antes do código ser passado para a fase de compilação.

O programador se comunica com o pré-processador inserindo *diretivas* em um código fonte de forma a facilitar a manutenção do programa. As diretivas para o pré-processador C podem ser reconhecidas pelo símbolo `#` na primeira coluna da linha onde ocorrem. Estas diretivas não são expressões C, de forma que as linhas onde elas ocorrem não são terminadas por ponto e vírgula.

5.1 A diretiva `#include`

A diretiva `#include` permite que um arquivo (geralmente com definições e declarações de protótipos) possa ser incluído em um módulo. Por convenção, tais arquivos — chamados de *arquivos de cabeçalho* — recebem a extensão `.h` (de *header*).

Já foi mencionado que a linguagem C oferece um conjunto de rotinas de suporte, das quais algumas já foram analisadas — `printf`, `malloc` e `free`, por exemplo. Para estas rotinas não é necessário que o usuário entre sempre com os protótipos de cada função que ele irá utilizar: esta informação já está contida em arquivos de cabeçalho definidos pelo compilador. Por exemplo, protótipos para rotinas de

alocação dinâmica estão em um arquivo de cabeçalho do sistema de nome `stdlib.h`. Assim, o exemplo da Seção 3.8 pode ser reescrito como:

```
/*
 * Exemplo de alocação dinâmica
 */
#include <stdlib.h>      /* prototipos das rotinas padrão */

main() {
    unsigned int n;      /* tamanho em inteiros da área */
    int *area;           /* apontador para início da área */

    /* obtém valor de n */
    ...

    /* obtém espaço da área livre */
    area = calloc(n, sizeof(int));
    if (area == 0) {
        printf("Erro de alocação\n");
        return(1);
    }

    /* trabalha com a área alocada */
    ...

    /* dobra a área original */
    area = realloc(area, 2*n*sizeof(int));
    if (area == 0) {
        printf("Erro de expansão\n");
        return(2);
    }

    /* continua trabalho com a área alocada */
    ...

    /* libera a área pré-alocada */
    free(area);

    /* tudo acabou bem... */
    return(0);
}
```

Neste caso, não foi preciso declarar os protótipos das rotinas de alocação dinâmica pois estes protótipos foram incluídos (como declarações globais para o módulo) através da diretiva *#include*. O arquivo incluído contém outras informações além dos protótipos das rotinas, mas isto fica transparente para o programador.

Observe que o nome do arquivo de cabeçalho foi incluído entre *<...>*. Isto indica que este arquivo é um arquivo de cabeçalho fornecido pelo compilador, e que será incluído a partir de um diretório padrão do sistema (geralmente o diretório */usr/include* em um sistema Unix).

Existe uma forma alternativa de inclusão que permite que o programador crie seus próprios arquivos de cabeçalho e os inclua em seus módulos. Uma vez que estes arquivos não podem ser escritos na área do compilador (em geral protegida contra escrita), é preciso indicar para o compilador que procure estes arquivos no próprio diretório onde está o módulo (ou outro diretório local do usuário). Neste caso, o nome do arquivo de cabeçalho deve ser incluído entre aspas.

Considere o exemplo da Seção 4.4. O programador deseja incluir em um arquivo de cabeçalho da aplicação a definição das estruturas e nomes de tipos de forma que isto não deva se repetir em cada módulo. Assim, ele cria um arquivo (por exemplo, *idade.h*) com o seguinte conteúdo:

```
/*
 *  idade.h
 *      - definicoes para a aplicacao de calculo de idade
 */
/* Define estruturas e nomes de tipos */
typedef enum sex {masculino, feminino} Sexo;

typedef struct data {
    int dia;
    int mes;
    int ano;
} Data;

typedef struct dados_pessoais {
    char nome[40];
    Data nascimento;
    Sexo genero;
} Pessoa;

/* prototipos: */
int calc_idade(Pessoa, Data);
```

```
Data le_hoje();
Pessoa *le_aluno();
```

No módulo com o programa, ele simplesmente inclui o arquivo de cabeçalho de sua aplicação:

```
#include "idade.h"

int main() {
    Data hoje;
    Pessoa *aluno_pt;
    int idade;

    /* obtém dados para hoje e aluno */
    hoje = le_hoje();
    aluno_pt = le_aluno();

    idade = calc_idade(*aluno_pt, hoje);

    /* apresenta resultado */
    printf("Idade de %s: %d\n",
           aluno_pt->nome,
           idade);

    return(0);
}
```

O uso da diretiva `#include` facilita a leitura do código C ao abstrair detalhes de definições para uma outra etapa e, principalmente, favorece a coerência entre módulos que devem compartilhar as mesmas definições.

5.2 A diretiva `#define`

Outra importante diretiva para o pré-processador C é a diretiva `#define`, que permite definir constantes simbólicas e macros que serão substituídas no código fonte durante a compilação. Com o uso desta diretiva torna-se mais simples manter o código envolvendo constantes correto. Ela também simplifica a compreensão do código ao usar nomes simbólicos que indicam o papel de constantes no módulo.

Considere o seguinte exemplo onde constantes são introduzidas diretamente no código:

```
/*
 *   Inicializacao do conteudo de um arranjo
 */
int main() {
    int notas[100];
    int aluno_ind;

    for (aluno_ind=0; aluno_ind<100; ++aluno_ind)
        notas[aluno_ind] = 100;

    ...
}
```

Nesta aplicação, até cem alunos podem estar na lista de notas, e a nota inicial de cada aluno é cem — a estratégia de avaliação deste professor é que o aluno pode perder pontos ao longo do curso. Caso a aplicação seja alterada para comportar duzentos alunos, uma substituição global de 100 por 200 iria introduzir erros no programa, uma vez que cada aluno iria iniciar o curso com uma nota duzentos!

Com o uso da diretiva *#define*, este trecho de programa poderia ser reescrito como

```
/*
 *   Inicializacao do conteudo de um arranjo
 */
/* definicao de constantes */
#define QTD_ALUNOS 100
#define NOTA_MAX 100

int main() {
    int notas[QTD_ALUNOS];
    int aluno_ind;

    for (aluno_ind=0; aluno_ind<QTD_ALUNOS; ++aluno_ind)
        notas[aluno_ind] = NOTA_MAX;

    ...
}
```

Após a expansão dos símbolos, este programa torna-se idêntico à versão anterior.

Agora torna-se claro que constantes são atribuídas às variáveis e quais são dimensões de arranjos. Para modificar o número de alunos, basta agora modificar a linha de definição

```
#define QTD_ALUNOS 200
```

que a aplicação poderá ser recompilada sem introdução de erros.

Um outro uso da diretiva `#define` é a definição de *macros*. Uma macro tem sintaxe de uso similar a uma chamada de função; entretanto, o código da macro é substituído no código fonte durante o pré-processamento. Macros não geram chamadas de funções, não têm variáveis na pilha e não fazem verificação de tipos de argumentos. Em geral, são utilizadas para substituir expressões complexas.

Considere a função `max` definida na Seção 2.5.1. Esta função tem um código que poderia ser expresso em uma linha com o uso do operador condicional. Entretanto, ela está restrita ao uso com variáveis inteiras. Para obter o maior valor entre duas variáveis do tipo `double`, outra função deveria ser escrita tendo exatamente o mesmo corpo — apenas o tipo de retorno e tipo dos argumentos seriam modificados.

Uma definição de macro pode simplificar este problema. A forma geral de definição de macros é

```
#define nome_macro(lista_argum) (corpo_macro)
```

O par de parênteses em torno do corpo da macro não é necessário, mas é usualmente incluído para evitar problemas de mudança de precedência após a expansão da macro no código fonte.

A macro para obter o máximo de dois valores poderia ser escrita como

```
#define max(a,b) (a<b ? b : a)
```

e utilizada da mesma forma que funções:

```
int i, j, k;
double x, y, z;
...
k = max(i,j);
z = max(x,y);
```

Após a fase de pré-processamento, o código efetivamente repassado para a fase de compilação seria:

```
int i, j, k;
double x, y, z;
...
k = (i<j ? j : i);
z = (x<y ? y : x);
```

A definição de uma macro pode se estender por mais de uma linha. Nestes casos, cada linha a ser continuada deve ser terminada por uma contrabarra (\). Por exemplo, a mesma macro `max` poderia ter sido definida como

```
#define max(a,b) \  
    ( a<b ? \  
      b : \  
      a )
```

O uso da diretiva `#define` também facilita a manutenção de código. Tais diretivas são usualmente incluídas como parte de arquivos de cabeçalho quando suas definições são compartilhadas entre diversos módulos. Por exemplo, a macro `max` exemplificada acima já é geralmente incluída em um arquivo padrão de cabeçalho, `macros.h`.

O pré-processador também entende uma diretiva `#undef`, que permite eliminar a definição de um identificador. Por exemplo,

```
#undef TRUE          /* esquece qualquer definicao anterior */  
#define TRUE 1       /* nova definicao */
```

5.3 Compilação Condicional

Em algumas situações, pode ser interessante incluir ou excluir alguns trechos de código em um programa — por exemplo, para incluir testes e mensagens de depuração durante o desenvolvimento do programa e excluí-los na versão final. Para programas de porte razoável, a manutenção manual deste tipo de trechos de programa pode se tornar uma tarefa complexa. Um mecanismo que pode facilitar esta tarefa é a utilização de diretivas de compilação condicional.

A diretiva básica para a compilação condicional é `#if ... #endif`:

```
#if expr_constante  
    /* codigo incluido quando expr_constante != 0 */  
    ...  
#else  
    /* codigo incluido quando expr_constante == 0 */  
    ...  
#endif
```

O trecho `#else` é opcional, podendo ser omitido. Um exemplo de uso destas diretivas é a verificação se uma constante já foi definida, como em

```

...
x = malloc(n);
#ifdef DEBUG
    printf("malloc: %d bytes alocados a partir de %p\n",
          n, x);
#endif
...

```

(A sequência de conversão `%p` apresenta uma variável apontador.) A função `printf` acima será invocada apenas quando o identificador `DEBUG` tiver sido previamente definido, como em

```
#define DEBUG
```

Observe que nem é necessário que um valor seja associado ao identificador neste caso; basta que ele esteja definido. Assim, durante a fase de desenvolvimento a definição acima seria incluída em módulos sendo depurados, sendo posteriormente removida para a geração do programa final.

A forma `#if defined(...)` ocorre tão frequentemente que há uma forma abreviada de diretiva, `#ifdef`. O exemplo acima poderia ser reescrito como

```

...
x = malloc(n);
#ifdef DEBUG
    printf("malloc: %d bytes alocados a partir de %p\n",
          n, x);
#endif
...

```

Um dos principais usos da compilação condicional é evitar a reinclusão de arquivos de cabeçalho. Em alguns casos, um arquivo de cabeçalho pode já incluir definições de outro arquivo de cabeçalho. A questão é: como evitar erros de redeclaração por causa de uma outra inclusão explícita de um arquivo já incluído implicitamente?

Por exemplo, suponha que a definição da estrutura `data` do exemplo de cálculo de idade estivesse em um arquivo `data.h`, pois é uma construção que será compartilhada por diversas aplicações:

```

/*
 * data.h
 */
typedef struct data {
    int dia;

```

```
        int mes;  
        int ano;  
    } Data;
```

A aplicação de manipulação de dados pessoais (tal como o cálculo da idade) requer esta definição para criar sua estrutura `dados_pessoais` e declarar os protótipos de funções trabalhando com a data no arquivo `idade.h`.

A primeira possibilidade é informar o usuário através da documentação do programa que, sempre que for incluir o arquivo `idade.h`, o arquivo `data.h` deve ser incluído antes, como em

```
#include "data.h"  
#include "idade.h"
```

Claramente, esta opção é incômoda, pois deixa a cargo do usuário a inclusão correta de arquivos. A segunda opção é incluir `data.h` dentro do arquivo `idade.h` de forma a garantir que a declaração estará presente quando a estrutura `dados_pessoais` for definida. O risco agora seria que o usuário também incluísse `data.h`, gerando erros de redefinição de estruturas.

A compilação condicional traz a solução para este caso. Quando um arquivo de cabeçalho é incluído pela primeira vez, ele pode definir um identificador associado apenas àquele arquivo. Quando se tenta incluir novamente o arquivo de cabeçalho, um teste é realizado — caso o identificador já esteja definido, então o conteúdo do arquivo não é incluído.

No caso de `data.h`, o símbolo poderia ser por exemplo `_H_DATA`, e o conteúdo do arquivo seria

```
/*  
 * data.h  
 */  
#if ! defined(_H_DATA)  
#define _H_DATA  
  
typedef struct data {  
    int dia;  
    int mes;  
    int ano;  
} Data;  
  
#endif
```

e o conteúdo de `idade.h` seria

```

/*
 * idade.h
 *      - definicoes para a aplicacao de calculo de idade
 */
#if ! defined(_H_IDADE)
#define _H_IDADE

#include "data.h"

/* Define estruturas e nomes de tipos */
typedef enum sex {masculino, feminino} Sexo;

typedef struct dados_pessoais {
    char nome[40];
    Data nascimento;
    Sexo genero;
} Pessoa;

/* prototipos: */
int calc_idade(Pessoa, Data);
Data le_hoje();
Pessoa *le_aluno();

#endif

```

Com estas construções, qualquer combinação de inclusão dos arquivos de cabeçalho:

```
#include "idade.h"
```

ou

```
#include "idade.h"
#include "data.h"
```

ou

```
#include "data.h"
#include "idade.h"
```

funcionarão de forma idêntica, sem erros de redeclaração.

Além de `#if`, `#ifdef`, `#else` e `#endif`, as diretivas `#ifndef` (se não definido) e `#elif` (else-if) são suportadas.

5.4 Outros Recursos

O pré-processador suporta o operador `#` para permitir substituir na macro a grafia de um argumento. Por exemplo, o programa

```
#define path(prof,curso) \
    "/home/faculty/" #prof "/courses/" #curso

main() {
    printf ("path: %s\n",path(ricarte,prog));
}
```

iria resultar na seguinte saída quando executado:

```
path: /home/faculty/ricarte/courses/progc
```

No exemplo acima, o recurso de concatenação de strings adjacentes (outra tarefa desempenhada pelo pré-processador) é utilizado.

Concatenação é suportada através do operador `##`. Quando este operador é usado em uma macro entre dois outros símbolos, os símbolos são inicialmente expandidos e então o símbolo do operador e quaisquer espaços em volta dele são eliminados. Por exemplo, a macro

```
#define sport(a)  a ## bol
```

poderia ser usada para criar identificadores em um programa, como

```
int sport(fute), sport(basquete);
...
futebol = 1;          /* criado por sport(fute) */
basquetebol = 2;      /* criado por sport(basquete) */
...
```

Há também macros que são pré-definidas e que podem ser usadas em qualquer programa C, que são:

__LINE__ uma constante decimal contendo o número da linha atual no arquivo fonte;

__FILE__ um string com o nome do arquivo fonte que está sendo compilado;

__DATE__ um string com a data da compilação;

__TIME__ um string com a hora (hh:mm:ss) da compilação.

Por exemplo, considere o seguinte programa criado em um arquivo de nome `predefin.c`:

```
main() {
    printf ("%s:%d (%s %s)\n",
            __FILE__, __LINE__, __DATE__, __TIME__);
}
```

Após compilado e executado, este programa apresentaria o seguinte resultado:

```
predefin.c:2 (May 17 1995 13:27:29)
```

Outras diretivas do pré-processador incluem:

#error *string* causa a geração pelo compilador de uma mensagem de erro contendo *string*;

#line *constante* *arquivo* indica novos valores para a macro `__LINE__` (passa a ter o valor *constante*, que deve ser um número decimal) e, caso *arquivo* esteja presente, para a macro `__FILE__`;

#pragma *string* é um mecanismo de comunicação com recursos não padronizados oferecidos pelo compilador, e cujo comportamento depende de cada implementação.

Por exemplo, considere que o arquivo `predefin.c` do exemplo anterior seja modificado como se segue:

```
main() {
    #ifdef TST
    #   line 100 "arq_teste"
    #else
    #   error Esqueceu de definir TST!
    #endif
    printf ("%s:%d (%s %s)\n",
            __FILE__, __LINE__, __DATE__, __TIME__);
}
```

A linha de comando

```
$ cc predefin.c -o predefin
```

geraria a seguinte resposta do compilador:

```
"predefin.c", line 5.0: 1506-205 (S) Esqueceu de definir TST!
```

É possível definir um símbolo na linha de comando de compilação através do uso da chave `-D`. A linha de comando

```
$ cc -DTST predefin.c -o predefin
```

geraria um programa executável `predefin` que quando executado geraria a seguinte mensagem:

```
arq_teste:103 (May 17 1995 14:10:27)
```

5.5 Atividades

Atividade 1 Como os comandos do pré-processador poderiam ser utilizados para comentar um trecho de código C que contenha comandos e comentários?

Atividade 2 Implemente os exemplos do capítulo.

Atividade 3 Revise a implementação do programa de cálculo de idade da atividade do capítulo anterior para incorporar os arquivos de cabeçalho. Modularize o seu código de forma que rotinas que manipulem exclusivamente operandos do tipo `Data` estejam em um módulo `data.c`, e estas rotinas sejam utilizadas no programa de cálculo de idade.

Capítulo 6

Funções em Bibliotecas

Nos capítulos anteriores foram apresentadas algumas funções da biblioteca padrão C tais como `printf` e as rotinas de alocação dinâmica de memória. Outras rotinas serão apresentadas neste capítulo.

É importante que se observe que informações sobre estas e outras funções podem ser obtidas a partir da referência *on-line* (seção 3 de `man` em um ambiente Unix com interação texto, `xman` em um ambiente Xwindows ou opção `help` em um ambiente em computador pessoal). Estas informações incluem tipo de retorno, número e tipos de argumentos, e quais arquivos de cabeçalhos devem ser incluídos para que se possa usar a função.

6.1 Entrada e Saída de Dados

Rotinas de entrada e saída estão usualmente associadas ao arquivo de cabeçalho `stdio.h`. Em geral, este arquivo deve ser incluído no arquivo fonte usando estas rotinas. Em alguns casos, estas “rotinas” são na verdade macros que estão definidas neste arquivo de cabeçalho.

6.1.1 Interação com Dispositivos Padrão

Os dispositivos padrão de interação com o usuário são o teclado (dispositivo de entrada padrão) e a tela do monitor (dispositivo de saída padrão). C suporta rotinas para acessar diretamente estes dispositivos.

A rotina básica de entrada de dados lê um caracter do teclado, retornando seu valor ASCII. Esta rotina é

```
#include <stdio.h>
int getchar();
```

A descrição acima deve ser lida da seguinte forma: o arquivo de cabeçalho `stdio.h` tem que ser incluído no arquivo fonte que for usar esta rotina `getchar`, e esta rotina não tem nenhum argumento e retorna um valor inteiro.

O seguinte exemplo ilustra o uso de `getchar`:

```
#include <stdio.h>

main() {
    int ch;

    ch=getchar();
    printf("valor ASCII de %c = %d (hexa %x)\n", ch, ch, ch);
}
```

Este programa, quando executado, irá aguardar que o usuário entre algum caracter via teclado. A saída será uma indicação de qual caracter foi obtido ao longo com seu valor em representação decimal e hexadecimal.

A rotina correspondente a `getchar` para a apresentação de um caracter na tela é `putchar`

```
#include <stdio.h>
int putchar(char);
```

Por exemplo, o programa anterior poderia ser estendido de forma a que um *prompt* (tal como `>`) indicasse ao usuário que uma entrada de dados é aguardada:

```
#include <stdio.h>

main() {
    int ch;

    putchar('>');
    ch=getchar();
    printf("valor ASCII de %c = %d (hexa %x)\n", ch, ch, ch);
}
```

É possível entrar um string a partir do teclado usando a rotina `gets`,

```
#include <stdio.h>
char *gets(char *);
```

Esta rotina lê um string do teclado armazenando-o no arranjo de caracteres apontado por seu argumento. A entrada do string é terminada pela tecla `ENTER`, que é substituído internamente pelo caracter `NUL`. O valor de retorno é o endereço do argumento se tudo correu bem ou o apontador nulo se houve alguma condição de erro. Por exemplo,

```
#include <stdio.h>

main() {
    char strch[60];
    char *pch;

    putchar('>');
    pch=gets(strch);
    if (pch != 0)
        printf("string aceito foi: %s\n", pch);
    else
        printf("Erro na obtencao do string!\n");
}
```

Este programa aceita um string via teclado, armazenando-o no arranjo de caracteres `strch`. Observe que o programador deve reservar o espaço que será utilizado para guardar o string. Aceitando o string, este será ecoado para a tela. Caso contrário (por exemplo, a entrada é o caracter de fim de arquivo `^D`), a mensagem de erro será apresentada.

A rotina de saída com formatação de valores, `printf`, já foi apresentada anteriormente. Há também uma rotina para a entrada de dados formatados, que é `scanf`:

```
#include <stdio.h>

int scanf(char *format, lista_enderecos);
```

O primeiro argumento de `scanf` é um string contendo sequências de conversão de formatos, seguindo o que já foi especificado em `printf`. Caracteres que não sejam parte de uma sequência de conversão no string de formato indica entrada que deve ser ignorada. O valor de retorno é o número de conversões realizadas com sucesso.

Por exemplo, para ler um valor inteiro do teclado diretamente para uma variável inteira, o seguinte programa poderia ser usado:

```
#include <stdio.h>

main() {
    int value;

    scanf("%d",&value);
    printf("numero aceito foi: %d\n", value);
}
```

Observe que o *endereço* da variável é o argumento de `scanf`.

6.1.2 Interação com Arquivos

Os dispositivos padrão de entrada e saída são apenas casos especiais de *arquivos*. Um arquivo é uma abstração que permite acessar convenientemente dispositivos externos (E/S), dos quais teclado e tela são casos particulares.

Usualmente, para trabalhar com um arquivo, o arquivo deve ser inicialmente aberto. A abertura indica ao sistema operacional que o arquivo será manipulado, de forma que informação que agiliza o acesso ao arquivo é mantida em memória. Após aberto, um arquivo é acessado através de um *descriptor* que basicamente indica onde o sistema mantém a informação sobre o arquivo.

Após aberto, operações de escrita e leitura podem ser efetuadas sobre o arquivo. Para que estas operações de transferência de dados tenham sucesso, é preciso que haja a permissão adequada para a operação. Por exemplo, um teclado seria um dispositivo que não aceita saída de dados (escrita), mas apenas entrada (leitura).

Encerradas as operações sobre um arquivo, ele deve ser fechado. Isto permite que o sistema operacional libere o espaço ocupado pelas informações sobre o arquivo para que este espaço possa ser reocupado na abertura de outros arquivos. Esta liberação é importante uma vez que sistemas operacionais limitam a quantidade de arquivos que podem ser abertos simultaneamente devido a limitações de espaço para estes dados.

Em C, a informação sobre um arquivo é mantida em uma estrutura interna, definida em `stdio.h`. Este arquivo de cabeçalho também define um nome de tipo `FILE` associada a esta estrutura, e o que o usuário usa para acessar arquivos são variáveis apontadores para este tipo, `FILE *`. Tais variáveis são usualmente chamadas de *manipuladores de arquivo*.

Para abrir um arquivo, a rotina `fopen` é invocada recebendo dois argumentos, um string com o nome do arquivo e outro string com o modo de acesso. O valor de retorno é o manipulador alocado para o arquivo aberto:

```
#include <stdio.h>
FILE *fopen(char *, char *);
```

Caso o arquivo não possa ser aberto, o apontador nulo é retornado.

O string com o modo de acesso pode conter os caracteres `r` (leitura), `w` (escrita), `a` (escrita ao final — *append*), e `b` (acesso em modo binário).

Para fechar um arquivo previamente aberto, a rotina `fclose` pode ser usada. Ela recebe como argumento o manipulador do arquivo e não retorna nenhum valor:

```
#include <stdio.h>
void fclose(FILE *);
```

Por exemplo, para abrir um arquivo chamado `teste.dat` em modo de escrita e leitura de dados binários, o seguinte segmento de código poderia ser utilizado:

```
#include <stdio.h>

int main() {
    FILE *fh;

    /* abre arquivo */
    fh = fopen("teste.dat", "rb");
    if (fh == 0) {
        printf ("Erro de abertura de arquivo\n");
        return(1);
    }

    /* opera com arquivo */
    ...

    /* fecha arquivo */
    fclose(fh);
    return(0);
}
```

Os arquivos referentes a teclado e tela são abertos automaticamente pelo sistema quando se inicia a execução de um programa. Eles podem ser acessados através dos manipuladores `stdin` (entrada padrão, teclado) e `stdout` (saída padrão, tela). Além destes dois arquivos, um terceiro arquivo padrão também é iniciado pelo sistema — o arquivo padrão de mensagens de erro, `stderr`, também associado à tela.

Há rotinas de acesso a arquivos similares às aquelas descritas para a interação com teclado e tela. (Na verdade, aquelas rotinas são criadas a partir destas.) Por exemplo, para obter um caracter de um arquivo pode-se utilizar a rotina `getc`:

```
#include <stdio.h>
int getc(FILE *);
```

Assim, `getchar` é equivalente a `getc(stdin)`. Há também uma rotina `fgetc` que tem o mesmo formato. A diferença entre elas é que `getc` é uma macro enquanto que `fgetc` é uma função.

Observe que o valor de retorno de `getc` é um inteiro. Isto permite testar se o caracter obtido é EOF (definido em `stdio.h`), que é um valor que não é armazenável em um tipo `char`. Por exemplo, o seguinte programa para apresentar o conteúdo de um arquivo texto na tela não termina propriamente:

```

/*
 * Exemplo de programa "type" que nao funciona!
 */
#include <stdio.h>

main(int argc, char *argv[]) {
    char  ch;          /* deve ser int! */
    FILE *arq;

    /* testa se numero de argumentos correto */
    if (argc != 2) {
        printf("Uso: %s <nome arquivo>\n",argv[0]);
        return(1);
    }
    /* abre arquivo para leitura */
    arq = fopen(argv[1],"r");
    if (arq == 0) {
        perror(argv[1]);
        return(1);
    }
    /* apresenta arquivo caracter a caracter */
    do {
        ch = getc(arq);
        putchar(ch);
    } while (ch != EOF);

    /* finalizacoes */
    fclose(arq);
    return(0);
}

```

O program deste exemplo não termina porque `ch`, sendo do tipo `char`, nunca será igual a `EOF`. Mudando-se a declaração de `ch` para `int` resolverá o problema.

Observe também neste exemplo a utilização da rotina de impressão de erros do sistema, `perror`:

```
void perror(char *);
```

Quando algum recurso do sistema operacional (neste caso, o sistema de arquivos) é utilizado e um erro ocorre, esta rotina pode ser utilizada para detectar que tipo de erro ocorreu. O argumento de `perror` é um string que irá preceder a mensagem de erro do sistema. Por exemplo, suponha que o usuário tente acessar um arquivo

chamado `abobora` que não existe. Então a abertura de arquivo para leitura irá falhar, e a seguinte mensagem será apresentada:

```
abobora: No such file or directory
```

Para inserir um caracter em um arquivo, a rotina `putc` pode ser usada:

```
#include <stdio.h>
int putc(char, FILE *);
```

Acesso a dados formatados em arquivos pode ser realizado através das rotinas `fprintf` e `fscanf`,

```
#include <stdio.h>
int fprintf(FILE *, char *, ...);
int fscanf(FILE *, char *, ...);
```

A entrada e saída de dados binários são suportadas pelas rotinas `fread` (leitura) e `fwrite` (escrita):

```
#include <stdio.h>
int fread(void *aptr, int tam, int qtde, FILE *arq);
int fwrite(void *aptr, int tam, int qtde, FILE *arq);
```

Estas rotinas suportam a transferência de blocos de `qtde` elementos de tamanho `tam` bytes cada elemento entre o arquivo com manipulador `arq` e a área de memória cujo endereço inicial é `aptr`. O valor retornado é o número de elementos transferidos de fato.

Por exemplo, supondo que um arquivo `teste.dat` contém cem números inteiros armazenados em formato binário (isto é, os padrões de bits da representação inteira de cada número são armazenados no arquivo). O seguinte programa lê estes cem inteiros do arquivo, incrementa cada inteiro caso seu conteúdo seja diferente de 0, e escreve de volta os valores atualizados para o arquivo:

```
/*
 *  atualiza NELEM valores inteiros em um arquivo NOMARQ
 */
#include <stdio.h>

#define NELEM 100
#define NOMARQ "teste.dat"

int main() {
    FILE *fp;
```

```
int buffer[NELEM];
int i, qtde_lida;

/* abrir arquivo */
if ((fp = fopen(NOMARQ,"rw")) == 0) {
    perror(NOMARQ);
    return(1);
}

/* le dados */
qtde_lida = fread(buffer, sizeof(int), NELEM, fp);

/* atualiza */
for (i=0; i<qtde_lida; ++i)
    if (buffer[i] != 0)
        buffer[i]++;

/* escreve dados atualizados */
fwrite(buffer, sizeof(int), qtde_lida, fp);

/* finalizacoes */
fclose(fp);
return(0);
}
```

6.2 Manipulação de Strings

O arquivo de cabeçalho `string.h` contém a declaração de diversas funções para a manipulação de strings em C. Uma vez que strings são arranjos de caracteres, não seria possível (por exemplo) copiar o conteúdo de um string `s2` para um string `s1` simplesmente por atribuição,

```
s1 = s2;          /* copia o endereco! */
```

ou comparar o conteúdo de dois strings diretamente,

```
if (s1 != s2)     /* compara os enderecos! */
    ...
```

Entre as rotinas suportadas para manipular strings na biblioteca padrão, estão

```
#include <string.h>
char *strcat(char *s1, const char *s2);
char *strncat(char *s1, const char *s2, size_t n);
int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
char *strcpy(char *s1, const char *s2);
char *strncpy(char *s1, const char *s2, size_t n);
size_t strlen(const char *s);
char *strchr(const char *s, int c);
```

Nos protótipos acima, `size_t` é um nome de tipo para representar tamanhos correspondente a algum tipo inteiro definido no arquivo de cabeçalho `stddef.h`.

A função `strcat` concatena o string `s2` ao string `s1`. O arranjo associado ao endereço `s1` deve ter espaço suficiente para armazenar o resultado concatenado. A função `strncat` acrescenta no máximo `n` caracteres de `s2` a `s1`.

A função `strcmp` compara o conteúdo de dois strings. Se o string `s1` for igual ao string `s2`, o valor de retorno é 0. Se os strings são diferentes, o valor de retorno será negativo quando o string `s1` for lexicograficamente menor que `s2`, ou positivo caso contrário. A função `strncmp` compara no máximo até `n` caracteres.

A função `strcpy` copia o string `s2` (até a ocorrência do caracter `'\0'`) para o arranjo apontado por `s1`. `strncpy` copia no máximo até `n` caracteres. Se o string `s2` for maior que `n` caracteres, o string resultante não será terminado pelo caracter `'\0'`.

A função `strlen` retorna o comprimento do string `s`, sem incluir o caracter `'\0'`.

A função `strchr` retorna o apontador para a primeira ocorrência do caracter `c` no string `s`, ou o apontador nulo se o caracter não está presente no string.

Além destas funções, é interessante destacar que existe uma função `sprintf` (declarada em `stdio.h`) que permite formatar valores como `printf`, com a diferença que a saída formatada é colocada em um string ao invés de ser enviada para a tela. Seu protótipo é:

```
#include <stdio.h>
int sprintf (char *s, const char *format, ...);
```

É responsabilidade do programador garantir que o string apontado por `s` tem espaço suficiente para armazenar o resultado.

6.3 Interação com o Sistema Operacional

Uma das formas de interação de um programa C com o sistema operacional é através da invocação da função `system`, também parte da biblioteca padrão:

```
#include <stdlib.h>
int system(const char *s);
```

O valor de retorno corresponde ao status de saída na execução do processo que executa o comando, sendo um valor distinto de 0 na ocorrência de problemas.

O exemplo abaixo ilustra o uso de `system`. Este exemplo é uma extensão do exemplo `type` anterior, onde além de apresentar o conteúdo do arquivo o programa também atualiza o atributo com a data do arquivo. Para atualizar este atributo, o comando `touch` do sistema operacional é utilizado. Este comando faz com que a data e hora associadas ao arquivo especificado como argumento sejam a data e hora correntes; se o arquivo especificado não existe, ele é criado com conteúdo vazio. Neste exemplo, se o arquivo não existe o programa encerra execução sem criar o arquivo, pois o comando `touch` é executado após a tentativa de abrir o arquivo com a função `fopen`.

```
/*
 * exemplo uso de system: leitura com atualizacao de data
 */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    FILE *arq;
    int ch;
    char comando[60];

    /* testa se numero de argumentos correto */
    if (argc != 2) {
        printf("Uso: %s <nome arquivo>\n",argv[0]);
        return(1);
    }

    /* abre arquivo para leitura */
    arq = fopen(argv[1],"r");
    if (arq == 0) {
        perror(argv[1]);
        return(1);
    }

    /* atualiza data do arquivo */
    /*-- prepara comando */
    sprintf(comando, "touch %s", argv[1]);
```

```
/*-- executa */
system(comando);

/* apresenta arquivo caracter a caracter */
do {
    ch = getc(arq);
    putchar(ch);
} while (ch != EOF);

/* finalizacoes */
fclose(arq);
return(0);
}
```

A outra forma de interação de um programa C com o sistema operacional é através da invocação de chamadas do sistema (*system calls*). O sistema operacional oferece um conjunto de serviços acessível através desta interface de rotinas que podem ser diretamente invocadas a partir de programas C.

Em geral, muito da funcionalidade das chamadas do sistema já está coberta em termos de rotinas equivalentes da biblioteca padrão. Por exemplo, é parte desta interface o conjunto de rotinas para manipulação de arquivos — *open*, *creat*, *close*, *read* e *write*. Estas rotinas são acessadas pelas funções C da biblioteca padrão que suportam a funcionalidade de entrada e saída para terminais e arquivos.

As rotinas do conjunto de chamadas do sistema são em geral extremamente dependentes de cada sistema operacional. No contexto de Unix há um esforço de padronização deste nível de chamadas, que é o padrão POSIX 1003.1 (*Portable Operating System* — o “IX” é uma referência não explícita a Unix). Entretanto, o padrão cobre aspectos básicos, sendo que há diversos grupos — tais como OSF (*Open Software Foundation*, consórcio liderado por IBM, DEC e HP) e UI (*Unix International*, consórcio liderado pela AT&T) — que buscam oferecer suas próprias extensões ao padrão, de forma que não há ainda uma interface única para todos os serviços providos por um sistema operacional.

Se portabilidade é importante para sua aplicação, o uso de chamadas do sistema deve ser evitado e, se realmente necessário, as chamadas utilizadas deveriam ser restritas ao conjunto POSIX.

6.4 Exemplo de Aplicativo

Além das rotinas da biblioteca padrão e de suporte, há diversos aplicativos que são diretamente suportados pela linguagem C. Estes aplicativos são suportados em geral através de bibliotecas que podem ser ligadas ao código da aplicação,

como ocorre com rotinas da biblioteca padrão — a diferença é que neste caso estas bibliotecas de aplicativos devem ser explicitamente indicadas para o compilador (por exemplo, através da chave `-l<lib>` para o comando `cc`, onde `<lib>` é uma indicação para a biblioteca do aplicativo).

Em Unix, por exemplo, há uma biblioteca de rotinas que suporta a funcionalidade básica de um gerenciador de base de dados. Esta biblioteca, `dbm`, inclui rotinas que permitem:

- iniciar (criar ou reabrir) e fechar uma base de dados;
- armazenar registros na base de dados, onde registros têm uma estrutura na forma `[chave, dados]`;
- buscar registros na base de dados, seja a partir de sua chave ou sequencialmente;
- remover um registro da base de dados.

Como pode se observar, `dbm` é uma biblioteca de funções de base de dados. Estas funções são disponíveis para o programador que necessita criar e manipular uma base de dados organizada através de uma função hash.

O uso convencional do `dbm` é através do armazenamento de pares chave/dado em um arquivo de dados. Cada chave deve ser única e associada a somente um item de dado.

Definições necessárias ao gerenciador estão incorporadas no arquivo de cabeçalho `dbm.h`. O tipo de dado `datum` define o componente básico de um registro, sendo este componente uma estrutura C na forma

```
typedef struct {  
    char *dptr;  
    int   dsize;  
} datum;
```

Esta estrutura permite chaves e itens de dado de tamanhos arbitrários. Um registro da base de dados é composto por dois elementos do tipo `datum` `[chave, dados]`.

As rotinas incorporadas à biblioteca do gerenciador são as seguintes:

dbminit inicia a base de dados.

Argumentos: nome da base de dados (`char *`).

Retorno: código de erro (`int`).

store armazena registro.

Argumentos: registro `[chave, dados]` (`datum, datum`).

Retorno: código de erro (`int`).

fetch busca um registro.

Argumentos: elemento com a chave do registro a ser buscado (*datum*).

Retorno: elemento com dados (*datum*).

delete remove um registro.

Argumentos: elemento com a chave do registro a ser removido (*datum*).

Retorno: código de erro (*int*).

firstkey recupera a chave do primeiro registro armazenado.

Argumentos: nenhum.

Retorno: elemento com a chave do registro (*datum*).

nextkey recupera a chave do registro seguinte ao registro dado.

Argumentos: elemento com a chave do registro conhecido (*datum*).

Retorno: elemento com a chave do registro seguinte ao registro dado como argumento (*datum*).

dbmclose fecha a base de dados.

Argumentos: nenhum.

Retorno: código de erro (*int*).

Rotinas que retornam um inteiro indicam erro através de valores negativos; sucesso é indicado por um valor de retorno 0. Por outro lado, rotinas que retornam um *datum* indicam erro setando o valor de *dptr* igual a 0 na estrutura retornada.

Para ligar programas C com a biblioteca *dbm*, a chave de compilação *-ldb* deve ser incluída. Para executar os programas, *dbm* requer que os arquivos correspondente à base de dados já existam. Para criar os arquivos para uma base de dados de nome *xxx*, pode-se utilizar o seguinte comando Unix:

`touch xxx.dir xxx.pag.`

O seguinte exemplo ilustra o uso desta biblioteca para armazenar dados em um arquivo com nomes e telefones:

```
/*
 * Armazena nome e telefone em um arquivo dbm "agenda"
 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <dbm.h>          /* define datum (typedef) */

#define ARQDBM "agenda"  /* nome da base de dados */
```

```
int main(int argc, char *argv[])
{
    int erro;
    datum dbname, dbfone;          /* estruturas para dbm */
    char nome[60], fone[20];        /* buffers para dados */
    char comando[60];               /* buffer para comando */

    /* ENTRADA DE DADOS */
    if (argc > 3) {
        /* numero invalido de argumentos */
        printf("Uso: %s [nome] [fone]\n",argv[0]);
        return(1);
    }
    else {
        if (argc == 1) {
            /* nenhum argumento fornecido */
            printf("\tNome: ");
            gets(nome);
            printf("\tFone: ");
            gets(fone);
        }
        else if (argc == 2) {
            /* nome foi fornecido em argv[1] */
            strcpy(nome,argv[1]);
            printf("\tFone: ");
            gets(fone);
        }
        else {
            /* nome e fone fornecidos */
            strcpy(nome,argv[1]);
            strcpy(fone,argv[2]);
        }
    }

    /* CONVERSAO DOS DADOS PARA ESTRUTURA DATUM */
    /* incluindo o ultimo '\0' nos strings */
    dbname.dptr = nome;
    dbname.dsize = strlen(nome)+1;
    dbfone.dptr = fone;
    dbfone.dsize = strlen(fone)+1;
```

```

/* ARMAZENA NA BASE DE DADOS */
/* -- prepara arquivos internos de dbm */
sprintf(comando,"touch %s.pag %s.dir", ARQDBM, ARQDBM);
system(comando);
/* -- abre base de dados */
erro = dbmopen(&ARQDBM, ARQDBM, 0);
if (erro != 0) {
    /* imprime mensagem de erro */
    perror("Abre agenda");
    return(1);
}
/* -- armazena dados na agenda */
erro = store(dbnome,dbfone);
if (erro != 0) {
    /* imprime mensagem de erro */
    perror("Armazena em agenda");
    return(1);
}
/* -- fecha base de dados */
erro = dbmclose(&ARQDBM);
if (erro != 0) {
    /* imprime mensagem de erro */
    perror("Fecha agenda");
    return(1);
}

/* -- tudo certo, saida sem erro */
return(0);
}

```

6.5 Atividades

Atividade 1 Implemente e teste a versão correta do programa `type`.

Atividade 2 Implemente o programa desenvolvido nas atividades do Capítulo 1 que apresenta o número de linha e de caracteres em um arquivo cujo nome é passado na linha de comando.

Atividade 3 Implemente um programa de cópia de arquivos que use as rotinas de leitura e escrita binária. O programa recebe os nomes dos arquivos fonte e destino através da linha de comando.

Atividade 4 Implemente um programa que receba um caracter do usuário e apresente sua representação binária. O programa deve requisitar o caracter interativamente e repetir esta operação até o momento em que o caracter `^D` for entrado.

Atividade 5 Estenda o exemplo da agenda `dbm` de forma que o usuário possa

- recuperar dados já armazenados a partir de um nome;
- listar o conteúdo completo da agenda;
- atualizar telefone de uma pessoa;
- remover uma pessoa e seus dados da agenda.

Capítulo 7

Introdução a C++

C++ é uma extensão da linguagem de programação C. As extensões de C++ sobre C foram primeiramente introduzidas por Bjarne Stroustrup em 1980 nos Laboratórios Bell de New Jersey. Inicialmente, a linguagem era chamada “C com classes”, mas o nome foi alterado para C++ em 1983.

A motivação para o desenvolvimento de C++ foi *complexidade*. Grandes sistemas implementados com a linguagem C, na ordem de 25000 a 100000 linhas de código, são difíceis de controlar ou mesmo entender sua totalidade. C++ surgiu para permitir que esta barreira seja quebrada. O objetivo de C++ é permitir que programadores possam gerenciar e compreender programas maiores e mais complexos.

A maior parte das adições introduzidas por Stroustrup suportam a programação orientada a objetos. Algumas das características de C++ foram inspiradas em outra linguagem, Simula67. Apesar de não ser exatamente uma linguagem orientada a objetos, Simula67 suportava diversos dos conceitos de abstração que são fundamentais para a programação orientada a objetos. A primeira linguagem orientada a objetos com repercussão significativa foi Smalltalk, desenvolvida no final da década de 70. Entretanto, seu uso foi e continua sendo muito restrito — C++ é a linguagem adotada pela maior parte de empresas e centros desenvolvendo software em grande escala.

O termo *orientação a objetos* pressupõe uma organização de software em termos de coleção de objetos discretos incorporando estrutura e comportamento próprios. Esta abordagem de organização é essencialmente diferente do desenvolvimento tradicional de software, onde estruturas de dados e rotinas são desenvolvidas de forma apenas fracamente acopladas.

Neste capítulo, as primeiras noções de orientação a objetos serão introduzidas. Esta breve visão geral do paradigma permitirá entender melhor os conceitos associados à programação orientada a objetos e, em particular, às construções da linguagem C++, que serão apresentadas na sequência. Há muito mais em C++

do que será coberto neste material introdutório — considere este capítulo apenas como um aperitivo para C++.

7.1 Programação Orientada a Objetos

Um *objeto* é uma entidade do mundo real que tem uma identidade, uma estrutura e um comportamento. Objetos podem representar entidades concretas (um arquivo no meu computador, uma bicicleta) ou entidades conceituais (uma estratégia de jogo, uma política de escalonamento em um sistema operacional).

A estrutura de um objeto é representada em termos de *atributos*. O comportamento de um objeto é representado pelo conjunto de *operações* que podem ser executadas sobre o objeto. Objetos com a mesma estrutura e o mesmo comportamento são agrupados em uma mesma *classe*. Uma classe é uma abstração que descreve propriedades do objeto que são importantes para uma aplicação e esconde o resto. Cada classe descreve um conjunto (possivelmente infinito) de objetos individuais. Cada objeto é dito ser uma *instância* de uma classe.

Polimorfismo significa que a mesma operação pode se comportar de forma diferente em classes diferentes. Por exemplo, a operação *move* quando aplicada a uma janela de um sistema de interfaces tem um comportamento distinto do que quando aplicada a uma peça de um jogo de xadrez. Um *método* é uma implementação específica de uma operação para uma certa classe.

Polimorfismo também implica que uma operação de uma mesma classe pode ser implementada por mais de um método. O usuário não precisa saber quantas implementações existem para uma operação, ou explicitar qual método deve ser utilizado: a linguagem de programação deve ser capaz de selecionar o método correto a partir do nome da operação, classe do objeto e argumentos para a operação. Desta forma, novas classes podem ser adicionadas sem necessidade de modificação de código já existente, pois cada classe apenas define os seus métodos e atributos.

Herança é o mecanismo do paradigma de orientação a objetos que permite compartilhar atributos e operações entre classes baseada em um relacionamento hierárquico. Uma classe pode ser definida de forma genérica e depois refinada sucessivamente em termos de *subclasses* ou *classes derivadas*. Cada subclasse incorpora, or *herda*, todas as propriedades de sua *superclasse* (ou *classe base*) e adiciona suas propriedades únicas e particulares. Assim as propriedades da classe base não precisam ser repetidas em cada classe derivada. Esta capacidade de fatorar as propriedades comuns de diversas classes em uma superclasse pode reduzir dramaticamente a repetição de código em um projeto ou programa, sendo uma das principais vantagens da abordagem de orientação a objetos.

A abordagem de orientação a objetos favorece a aplicação de diversos conceitos

considerados fundamentais para o desenvolvimento de bons programas, tais como abstração e encapsulação. Tais conceitos não são exclusivos desta abordagem, mas são suportados de forma melhor no desenvolvimento orientado a objetos do que em outras metodologias.

Técnicas de orientação a objetos promovem compartilhamento em diversos níveis distintos. Herança de estrutura de dados e comportamento permite que estruturas comuns sejam compartilhadas entre diversas classes derivadas similares sem redundância. O compartilhamento de código usando herança é uma das grandes vantagens da orientação a objetos. Ainda mais importante que a economia de código é a clareza conceitual de reconhecer que operações diferentes são na verdade a mesma coisa, o que reduz o número de casos distintos que devem ser entendidos e analisados.

O desenvolvimento orientado a objetos não apenas permite que a informação dentro de um projeto seja compartilhada como também oferece a possibilidade de reaproveitar projetos e código em projetos futuros. As ferramentas para alcançar este compartilhamento, tais como abstração, encapsulação e herança, estão presentes na metodologia; uma estratégia de reuso entre projetos é a definição de bibliotecas de elementos reusáveis. Entretanto, orientação a objetos não é uma fórmula mágica para alcançar reusabilidade; para tanto, é preciso planejamento e disciplina para pensar em termos genéricos, não voltados simplesmente para a aplicação corrente. Passar a programar usando os mecanismos de orientação a objetos ao invés de métodos tradicionais (como programação estruturada) requer em geral um processo de reeducação de programadores.

7.2 Particularidades de C++

C++ tem muito a ver com a linguagem C, da qual ela é derivada. Os tipos e formatos de expressões, operadores e comandos da linguagem são os mesmos. Arranjos, *strings*, apontadores e funções são também definidos e manipulados como em C. Os tipos de dados básicos em C++ são os mesmos que são definidos pelo padrão ANSI para a linguagem C.

C++ introduz uma construção para facilitar a representação de comentários de uma única linha — a *barra dupla* `//`, que pode começar em qualquer posição de uma linha e considera como comentário o restante da linha. Apesar de o padrão C de comentários ainda ser válido, recomenda-se a adoção exclusiva da forma `//`.

Outra construção que é nova em C++ é o operador de escopo `::`. Não há correspondente para este operador em ANSI-C: ele permite acessar uma variável global (com o uso do prefixo `::`) mesmo que exista uma variável local com o mesmo nome. Deve-se observar que, apesar de disponível em C++, este tipo de construção não é recomendado; o próprio uso de variáveis globais não é conside-

rado boa prática de programação. Entretanto, caso seja realmente necessário, o mecanismo de acesso a variáveis globais está disponível.

Protótipos, opcionais em ANSI-C, são mandatórios em C++. Em ANSI-C, duas funções devem ter nomes obrigatoriamente únicos. Em C++, uma vez que o mecanismo de prototipagem permite identificar uma função por seu nome e seus argumentos, é possível repetir nome de funções — é o mecanismo de *sobrecarga*. Observe que o tipo de retorno da função não é um critério de seleção para determinar qual função será chamada. A seleção da função é feita em tempo de compilação, e não em tempo de execução.

7.2.1 Entrada e Saída

C++ também não oferece operações de entrada e saída como parte da linguagem em si. Ao invés disto, ela oferece uma biblioteca que adiciona as funções de entrada e saída de forma elegante, usando o conceito de *streams*. O arquivo de cabeçalho que contém definições necessárias para o uso de streams é `iostream.h`.

O próximo exemplo mostra, ainda que de forma muito simples, o uso dos métodos de entrada e saída. Algumas variáveis são definidas e enviadas para o dispositivo de saída padrão através do stream `cout`. Observe como, ao contrário do que acontece com `printf` em C, o programador não tem de dizer ao sistema que tipo de dado está sendo enviado para a saída.

```
#include <iostream.h>
#include <string.h>

main() {
    int index;
    float distance;
    char letter;
    char name[25];

    index = -23;
    distance = 12.345;
    letter = 'X';
    strcpy(name, "John Doe");

    cout << "The value of index is "    << index    << "\n";
    cout << "The value of distance is " << distance << "\n";
    cout << "The value of letter is "  << letter  << "\n";
    cout << "The value of name is "    << name    << "\n";
```

```

    index = 31;
    cout << "The decimal value of index is "
          << dec << index << "\n";
    cout << "The octal value of index is "
          << oct << index << "\n";
    cout << "The hex value of index is "
          << hex << index << "\n";
    cout << "The character letter is "
          << (char)letter << "\n";

    cout << "Input a decimal value --> ";
    cin  >> index;
    cout << "The hex value of the input is "
          << index << "\n";
}

```

O resultado da execução deste programa seria:

```

The value of index is -23
The value of distance is 12.345
The value of letter is X
The value of name is John Doe
The decimal value of index is   31
The octal value of index is    37
The hex value of index is     1f
The character letter is X
Input a decimal value --> 999
The hex value of the input is 3e7

```

O operador `<<`, chamado de *operador de inserção*, diz para o sistema para enviar a seguinte variável ou constante para a saída, mas deixa o sistema decidir como apresentar o dado. Observe a consistência que este operador apresenta: em uma mesma linha de código, ele é utilizado para apresentar um string de caracteres e números (inteiros, ponto flutuante), sem que o programador tenha que se preocupar qual a rotina que será chamada para a apresentação. O sistema é encarregado de decidir qual a rotina apropriada.

Além do stream `cout`, há também o stream `cin` que é usado para ler dados do dispositivo de entrada padrão. Neste caso, o operador `>>`, chamado de *operador de extração*, é utilizado. Como no uso de `cout`, os operadores `dec`, `oct` e `hex` podem selecionar a base de numeração adotada para a entrada; como usual, a base decimal é o padrão quando nada é especificado.

Há também um stream `cerr` pré-definido, que envia dados para o dispositivo de erros padrão. Estes três streams, `cout`, `cin` e `cerr` correspondem aos apontadores para stream `stdout`, `stdin` e `stderr` da linguagem C, e são automaticamente abertos e fechados pelo sistema.

7.2.2 Definição de Variáveis

Assim como em C, variáveis globais e estáticas em C++ são automaticamente inicializadas para o valor 0 quando declaradas caso um valor diferente não seja especificado. Variáveis automáticas, que são declaradas dentro do escopo de uma função, não são inicializadas pelo sistema.

O próximo exemplo ilustra algumas das particularidades de definição de variáveis em C++.

```
#include <iostream.h>

int index;

main() {
    int stuff;
    int &another_stuff = stuff;
    // another_stuff is a reference for stuff

    //index was initialized to zero
    stuff = index + 14;
    cout << "stuff has the value " << stuff << "\n";
    stuff = 17;
    cout << "another_stuff has the value "
         << another_stuff << "\n";

    // more_stuff is not automatically initialized
    int more_stuff = 13;

    cout << "more_stuff has the value "
         << more_stuff << "\n";

    for (int count = 3; count < 8; count++) {
        cout << "count has the value " << count << "\n";
        char count2 = count + 65;
        cout << "count2 has the value " << count2 << "\n";
    }
```

```
// goofy is automatically initialized to zero
static unsigned goofy;

    cout << "goofy has the value " << goofy << "\n";
}
```

O resultado da execução deste programa seria

```
stuff has the value 14
another_stuff has the value 17
more_stuff has the value 13
count has the value 3
count2 has the value D
count has the value 4
count2 has the value E
count has the value 5
count2 has the value F
count has the value 6
count2 has the value G
count has the value 7
count2 has the value H
goofy has the value 0
```

Este exemplo ilustra a possibilidade de se definir variáveis (como `more_stuff`, `count` e `goofy`) no meio de uma função. Em C, todas variáveis de uma função deviam ser definidas antes do primeiro comando; em C++, variáveis podem ser definidas mais próximas dos pontos onde elas serão efetivamente utilizadas. Tais variáveis têm escopo a partir do ponto onde são declaradas até o final do bloco onde foram definidas.

Observe o uso do operador `&` na declaração de variáveis, outra particularidade de C++. No exemplo, `another_stuff` é criada como uma *variável de referência* à `stuff`. Este exemplo coloca apenas uma ilustração deste recurso — seu uso mais prático é na passagem de argumentos para funções por referência.

O uso de enumeração e estrutura é similar ao uso em linguagem C, exceto por uma pequena diferença: não é necessário repetir a palavra chave (`enum` e `struct`, respectivamente) para declarar uma variável deste tipo composto. Por exemplo, se um tipo de enumeração foi previamente declarado como:

```
enum game_result {win,lose,tie};
```

então uma variável deste tipo pode ser declarada simplesmente como

```
game_result outcome;
```

ou seja, `outcome` é uma variável que pode apenas assumir os valores `win`, `lose` ou `tie`. Não é necessário usar `typedef`.

A definição de classes traz o maior potencial de C++, e este assunto será abordado em mais detalhes adiante. Neste momento, entretanto, é interessante notar que classes podem ser definidas e acessadas praticamente da mesma maneira que estruturas.

A forma tradicional (ANSI-C) de conversão de tipos, através do operador de molde, é também aceita em C++. Entretanto, uma nova forma é introduzida em C++: a conversão de forma equivalente à chamada de uma função. Por exemplo, a expressão

```
a = (int) b;
```

poderia ser reescrita como

```
a = int(b);
```

Observe que não é uma boa regra de programação misturar estas duas formas de conversão. Uma delas deve ser adotada e utilizada ao longo de todo a codificação de um programa.

7.2.3 Alocação Dinâmica

Em C, alocação dinâmica é tratada por rotinas da biblioteca padrão. Estas rotinas tornaram-se tão utilizadas que sentiu-se, durante o projeto de C++, que sua funcionalidade devia fazer parte da linguagem de maneira a melhorar sua eficiência. Para tanto, foram introduzidos os operadores `new` e `delete`, que são parte integrante de C++ assim como um operador de adição ou atribuição.

O exemplo abaixo ilustra o uso destes operadores.

```
#include <iostream.h>

struct date {
    int month;
    int day;
    int year;
};

main() {
    int index, *point1, *point2;
```

```

    point1 = &index;
    *point1 = 77;
    point2 = new int;
    *point2 = 173;
    cout << "The values are " << index << " " <<
        *point1 << " " << *point2 << "\n";
    point1 = new int;
    point2 = point1;
    *point1 = 999;
    cout << "The values are " << index << " " <<
        *point1 << " " << *point2 << "\n";
    delete point1;

float *float_point1, *float_point2 = new float;

    float_point1 = new float;
    *float_point2 = 3.14159;
    *float_point1 = 2.4 * (*float_point2);
    delete float_point2;
    delete float_point1;

date *date_point;

    date_point = new date;
    date_point->month = 10;
    date_point->day = 18;
    date_point->year = 1938;
    cout << date_point->month << "/"
        << date_point->day << "/"
        << date_point->year << "\n";
    delete date_point;

char *c_point;

    c_point = new char[37];
    delete c_point;
    c_point = new char[sizeof(date) + 133];
    delete c_point;
}

```

Observe que o operador `new` requer um modificador, o qual deve ser um tipo.

Neste exemplo, `point2` aponta para uma variável inteira que foi dinamicamente alocada e pode, a partir daquele momento, ser usada como uma variável normal.

Posteriormente neste exemplo `point1` também passa a apontar para uma variável dinamicamente alocada, e `point2` recebe este mesmo endereço. Observe que neste ponto o endereço a que `point2` originalmente se referia é perdido, de forma que não é possível desalocar `point2`. Por este motivo, o operador `delete` é apenas aplicado a `point1`. Após a aplicação do operador `delete`, o espaço usado pela variável dinâmica torna-se disponível para outras alocações; portanto, os apontadores liberados desta forma não devem ser referenciados depois.

Caso o argumento para o operador `delete` seja o apontador nulo, o operador nada fará. Efetivamente, este operador só pode ser aplicado a variáveis que tenham sido alocadas com o operador `new`. Se ele for aplicado a qualquer outro tipo de variável, então qualquer coisa pode acontecer — até mesmo um *crash* do sistema.

O exemplo também mostra o uso destes operadores com outros tipos de variáveis, neste caso `floats` e `structs`.

Finalmente, neste exemplo ilustra-se a utilização destes operadores para alocar e desalocar arranjos. Neste caso, aloca-se um arranjo de 37 caracteres e também um arranjo cujo tamanho é 133 bytes maior que o tamanho da estrutura `date`.

Note que é *muito* importante a indicação, quando for o caso, de que se está removendo um arranjo que foi dinamicamente alocado com `new`. Esta indicação é feita usando-se o modificador `[]` após o operador `delete`. A ausência do modificador não será acusada como erro nem em tempo de compilação nem em tempo de execução; porém, ela pode comprometer a correta execução de programas mais complexos do que o apresentado neste exemplo. Esta responsabilidade é do programador.

O resultado da execução deste programa é:

```
The values are 77  77  173
The values are 77  999  999
10/18/1938
```

As funções de C para o gerenciamento de memória dinâmica, `malloc()`, `calloc()` e `free()`, podem ainda ser utilizadas em C++. Entretanto, seu uso não deve ser misturado com os novos operadores introduzidos por C++, ou os resultados podem ser imprevisíveis. Novos programas devem utilizar exclusivamente os novos operadores, uma vez que estes são muito mais eficientes.

7.2.4 Argumentos de Funções

Passagem por referência permite passar uma variável para uma função e retornar as mudanças feitas pela função para o programa anterior. O mesmo efeito

pode ser obtido com apontadores em ANSI-C, mas o mecanismo de referência é mais claro e elegante.

O seguinte exemplo ilustra a combinação do mecanismo de referência e passagem de argumentos para funções.

```
#include <iostream.h>
#include <stdio.h>

void fiddle(int in1, int &in2);

main() {
    int count = 7, index = 12;

    cout << "The values are ";
    printf("%3d %3d\n", count, index);

    fiddle(count, index);

    cout << "The values are ";
    printf("%3d %3d\n", count, index);
}

void fiddle(int in1, int &in2) {
    in1 = in1 + 100;
    in2 = in2 + 100;
    cout << "The values are ";
    printf("%3d %3d\n", in1, in2);
}
```

Observe que o segundo argumento da função `fiddle` é precedido por `&`. A consequência é que, para este argumento, a variável “verdadeira” do programa anterior será usada; o normal, passagem por valor, é que apenas uma cópia do valor corrente da variável seja passada para a função. Este efeito explica o resultado obtido pela execução deste programa, apresentado a seguir.

```
The values are    7  12
The values are  107 112
The values are    7 112
```

C++ também permite a especificação de valores que argumentos de função assumirão caso a função seja chamada sem estes parâmetros — são os *parâmetros default*. O exemplo abaixo ilustra o uso deste mecanismo em uma função com

três argumentos, dois dos quais podem assumir valores *default*, mas pelo menos o primeiro deve estar sempre presente. No corpo da função, há chamadas para a função com um, dois e três argumentos.

```
#include <iostream.h>
#include <stdio.h>

int get_volume(int length, int width = 2, int height = 3);

main() {
    int x = 10, y = 12, z = 15;

    cout << "Some box data is " << get_volume(x, y, z) << "\n";
    cout << "Some box data is " << get_volume(x, y) << "\n";
    cout << "Some box data is " << get_volume(x) << "\n";

    cout << "Some box data is ";
    cout << get_volume(x, 7) << "\n";
    cout << "Some box data is ";
    cout << get_volume(5, 5, 5) << "\n";
}

int get_volume(int length, int width, int height) {
    printf("%4d %4d %4d  ", length, width, height);
    return length * width * height;
}
```

O resultado da execução deste programa deve ser:

```
Some box data is   10   12   15   1800
Some box data is   10   12    3    360
Some box data is   10    2    3     60
Some box data is   10    7    3    210
Some box data is    5    5    5    125
```

7.3 Classes

Uma definição de classe é basicamente uma definição de um tipo de dado. Uma classe contém um conjunto de bits representando um estado e um conjunto de operações que permitem modificar o estado.

Em geral, as operações são *públicas* e os dados internos da classe são *privados* — as únicas modificações possíveis nos dados são realizadas através das operações

que a classe deixa disponível para que outros usem. A informação que permanece escondida do usuário da classe está contida na *seção privativa* da classe. De fora da classe, é como se aqueles dados não existissem: eles não podem ser acessados ou modificados diretamente.

Funções declaradas dentro de uma classe são chamadas de *funções membros*, uma vez que elas são membros da classe da mesma forma que os dados declarados na classe. Funções membros são definidas da mesma forma que funções normais — a única diferença visível é que o nome da função é precedido pelo nome da classe, sendo os nomes separados por `::`. Uma outra diferença é que dados privados da classe podem ser acessados por elas, tanto para leitura como para alteração.

Em geral, a parte privativa da classe contém apenas dados, enquanto que a parte pública contém apenas declarações de funções. Entretanto, nada impede que dados sejam declarados também na parte pública e funções na parte privativa.

O exemplo a seguir apresenta, de forma muito simplificada, os conceitos apresentados acima. Neste programa, uma classe contendo um único dado — o inteiro `data_store` — é declarada, e todo acesso ao estado da classe (o valor deste inteiro) é restrito a suas duas funções membro, `set` e `get_value`. Estas duas funções são os *métodos* da classe, na terminologia de orientação a objetos apresentada.

```
#include <iostream.h>

// definicao da classe
class one_datum {
    int data_store;
public:
    void set(int in_value);
    int get_value(void);
};

// metodos
void one_datum::set(int in_value) {
    data_store = in_value;
}

int one_datum::get_value(void) {
    return data_store;
}

main() {
    one_datum dog1, dog2, dog3;
    int piggy;
```

```
    dog1.set(12);
    dog2.set(17);
    dog3.set(-13);
    piggy = 123;

    // dog1.data_store = 115;      ilegal em C++
    // dog2.data_store = 211;      ilegal em C++

    cout << "The value of dog1 is "
          << dog1.get_value() << "\n";
    cout << "The value of dog2 is "
          << dog2.get_value() << "\n";
    cout << "The value of dog3 is "
          << dog3.get_value() << "\n";
    cout << "The value of piggy is " << piggy << "\n";
}
```

O resultado da execução deste programa é

```
The value of dog1 is 12
The value of dog2 is 17
The value of dog3 is -13
The value of piggy is 123
```

Observe que, dentro da definição da classe, o que está presente na verdade são os protótipos dos métodos.

No início da função `main` é mostrado como objetos da classe definida são criados — da mesma forma que variáveis dos tipos padrão da linguagem.

O processo de chamar de um dos métodos da classe aplicado a um objeto da classe é conhecido na terminologia de orientação a objetos como *enviar uma mensagem*. A maneira de se enviar uma mensagem em C++ é ilustrada neste exemplo. Por exemplo, a linha onde se lê `dog1.set(12)` pode ser interpretada como “envie uma mensagem para o objeto `dog1` instruindo-o para setar (seu valor interno para) 12.” Repare como esta forma de acesso assemelha-se sintaticamente ao acesso de dados internos de uma estrutura: o nome do objeto, um ponto (`.`), e o nome do método.

O exemplo também ilustra, sob a forma de comentários, a “forma ilegal” de acesso aos dados internos dos objetos. Caso se tentasse compilar um programa com um comando daquela forma, um erro seria acusado pelo compilador C++.

7.3.1 Construtores e Destrutores

Construtores são basicamente funções de inicialização de uma classe, as quais são invocadas no momento em que objetos desta classe são criadas. Eles permitem inicializar campos internos da classe e alocar recursos que um objeto da classe possa demandar, tais como memória, arquivos, semáforos, soquetes, etc.

A função construtor de uma classe é reconhecida por ter o mesmo nome da classe. Por exemplo, um construtor para uma classe chamada `rectangle` seria o método `rectangle()` desta classe. Construtores podem receber argumentos, de forma que é possível sobrecarregar construtores.

Construtores podem ser usados para suportar a inicialização de valores internos da classe durante a declaração de objetos. Neste caso, os argumentos da função construtor são os valores que deverão ser inicializados para os dados do objeto. Quando este recurso for utilizado, é importante observar que construtores devem ser definidos de forma a cobrir todos as formas de inicialização desejadas — por exemplo, deve-se fornecer em geral um construtor sem argumentos para objetos não inicializados na declaração, ou que assumam valores default.

Destrutores realizam a função inversa: são funções invocadas quando um objeto está para “morrer”. Caso um objeto tenha recursos alocados, destrutores devem liberar tais recursos. Por exemplo, se o construtor de uma classe alocou uma variável dinamicamente com `new`, o destrutor correspondente deve liberar o espaço ocupado por esta variável com o operador `delete`.

A função destrutor de uma classe tem o mesmo nome da classe com um til (`~`) como prefixo. Por exemplo, o método destrutor para a classe `rectangle` seria chamado `~rectangle()`.

Quando um objeto é usado para inicializar outro, uma cópia bit a bit é feita. Em muitos casos, este mecanismo de cópia é perfeitamente adequado. Porém, há casos onde esta cópia simples pode ser perigosa; por exemplo, quando um objeto contém apontadores para áreas alocadas dinamicamente que não devem ser compartilhadas.

Para tais situações, C++ permite a definição de um *construtor de cópia*. Quando um construtor de cópia existe, a cópia por bits não é utilizada. A forma geral de declarar um construtor de cópia é

```
nome-classe (const nome-classe& obj) {  
    // definicao do construtor de copia  
    ...  
};
```

onde `obj` é uma referência para o objeto do lado direito da inicialização.

A forma de uso do construtor de cópia durante a inicialização de um objeto é, assumindo que exista uma classe `rectangle`,

```
rectangle x = y;
```

ou

```
rectangle x(y);
```

Em qualquer caso, uma referência para `y` seria passada para `obj`.

Observe que o construtor de cópia só é chamado automaticamente para inicializações. Um comando de atribuição no meio de uma função não ativa o construtor de cópia — se este for o efeito desejado, o operador `=` deve ser sobrecarregado.

7.3.2 Objetos e Funções

Uma vez que objetos são equivalentes a qualquer dado de outro tipo, é possível passar objetos como argumentos de uma função. Objetos são passados para função por valor, ou seja, uma cópia do objeto é feita para o parâmetro formal da função quando o objeto é passado como argumento. Da mesma forma, uma função pode ter como tipo de retorno uma classe, e assim retornar um objeto para a função anterior.

Quando um objeto é passado como um argumento para uma função, o construtor do objeto *não* é chamado — a cópia é feita bit a bit. Entretanto, quando a função termina, a cópia é um objeto que deixará de existir e, portanto, o destrutor para o objeto (se existente) será chamado. Quando este comportamento puder trazer problemas, o objeto deve ser passado por referência e não por valor.

Quando uma função membro é chamada, ela automaticamente recebe como argumento um apontador para o objeto que ativou o método; este apontador é chamado `this`. Quando uma função membro de uma classe acessa diretamente um dado privativo de um objeto, na verdade este apontador está sendo implicitamente utilizado para identificar o objeto. Algumas vezes, pode ser necessário ter que explicitar o uso deste apontador — por exemplo, quando se espera que a função membro retorne um objeto.

7.3.3 Sobrecarga de Operadores

Com relação a classes, há um aspecto muito relacionado à sobrecarga de funções — é a sobrecarga de operadores. Em outras palavras, é possível definir os símbolos usuais de operadores — tais como `+`, `*` ou `<<` — de forma a associá-los com métodos definidos para a classe.

A forma de implementar esta característica é através da definição de uma *função operador*. Uma função operador é geralmente uma função membro da classe. (Caso não o seja, ela deve ser uma função amiga, como explicado na próxima seção.) A declaração de uma função operador é da forma

```
tipo classe::operator# (lista-arg);
```

onde o símbolo `#` é substituído pelo operador que irá ser sobrecarregado.

Por exemplo, suponha que uma classe de nome `loc` seja definida com dois dados privativos, `longitude` e `latitude`. Um operador de adição para esta classe poderia ser definido como

```
loc loc::operator+(loc op2) {
    loc temp;

    temp.longitude = op2.longitude + longitude;
    temp.latitude = op2.latitude + latitude;
    return(temp);
}
```

A forma de uso para este operador seria então

```
loc loc1, loc2, newloc;
...
newloc = loc1 + loc2;
```

Observe que `operator+()` tem apenas um parâmetro, mesmo sobrecarregando um operador binário (+). O que ocorre é que o parâmetro do lado esquerdo do sinal + é passado implicitamente para a função operador usando o apontador `this`. O operando do lado direito é passado como o parâmetro `op2`. O fato de que o operando do lado esquerdo é passado usando `this` tem uma implicação importante: quando operadores binários são sobrecarregados, é o objeto à esquerda que gera a chamada para a função operador.

É importante notar que novos operadores não podem ser criados — apenas aqueles que existem podem ser sobrecarregados por uma classe. Do mesmo modo, a precedência de um operador não pode ser modificada.

7.4 Herança

O mecanismo de herança é o que diferencia a programação orientada a objetos da programação de Tipos Abstratos de Dados. Este é um dos conceitos mais importantes para a efetiva utilização de C++.

Para ilustrar como o mecanismo de herança é implementado em C++, considere o seguinte exemplo, onde uma classe para representar veículo é definida.

```
// vehicle.h
#ifndef _H_VEHICLE
```

```
#define _H_VEHICLE

class vehicle {
    int wheels;
    float weight;
public:
    void initialize(int in_wheels, float in_weight);
    int get_wheels(void);
    float get_weight(void);
    float wheel_loading(void);
};

#endif
```

Esta classe consiste de quatro métodos que manipulam os dois atributos relacionados ao número de rodas e peso do veículo. Observe que esta definição é genérica o suficiente para representar estes aspectos tanto para uma bicicleta quanto para um avião a jato. Esta é uma característica de classes base — elas *abstraem* as propriedades de um grupo de classes.

A questão agora é: como representar (por exemplo) carros e caminhões? Eles são tipos de veículos, e uma declaração de classe que simplesmente repetisse as propriedades de veículos para carros ou caminhões iria simplesmente perder esta informação. Pior ainda: se a classe de veículos fosse atualizada para introduzir uma nova propriedade (atributo ou método), esta mudança não seria refletida nas classes de carros ou caminhões. Este tipo de mudança ocorre muitas vezes em projetos de grande porte, quando há revisões de especificação para partes do projeto; em geral, é muito difícil controlar a propagação das modificações, principalmente quando diversos grupos de trabalho estão envolvidos no desenvolvimento.

Em C++, como em outras linguagens de programação orientadas a objetos, há um mecanismo para conectar a definição da classe derivada com a classe base. Observe abaixo como a classe para carros é definida, assumindo-se que a classe `vehicle` foi definida como acima.

```
// car.h
#ifndef _H_CAR
#define _H_CAR

#include "vehicle.h"

class car : public vehicle {
    int passenger_load;
```

```
public:
    void initialize(int in_wheels,
                   float in_weight,
                   int people = 4);
    int passengers(void);
};
#endif
```

O mecanismo para indicar que a classe `car` é derivada da classe `vehicle` é dado pela sintaxe : `public` na linha de declaração da classe, como em

```
class Derived : public Base {
    ...
};
```

que poderia ser lido como *Derived é-um-tipo-de Base*. Isto indica que a classe `Derived` é composta por toda a informação que está contida na classe `Base`, além de sua própria informação.

Na definição da classe `car`, indica-se que carro tem a mesma informação que um veículo, além de informação adicional sobre o número de passageiros que ele comporta. Note também que a classe `car` define um método de nome `initialize`, que era também um método da classe `vehicle`. Desta forma, quando `initialize` for ativado para um carro, a nova versão é que será ativada — para outros veículos, a antiga versão continua valendo.

7.4.1 Herança de Construtores e Destrutores

É possível que classes base e derivada tenham construtores e destrutores associados. Uma questão que pode surgir é, durante a construção de um objeto derivado, qual a ordem de chamada dos construtores? Similarmente, qual a ordem de chamada de destrutores quando o objeto é removido?

Quando um objeto de uma classe base é criado, primeiro o construtor da classe base (se existe um construtor) é invocado, e depois o construtor da classe derivada é executado. Por outro lado, quando um objeto é removido, a ordem inversa é obedecida: primeiro o destrutor da classe derivada é executado, e apenas então o destrutor da classe base é invocado.

Este procedimento é o mesmo qualquer que seja o nível de derivação: na construção, de base para derivada, e na destruição, de derivada para base. O mesmo é válido mesmo no caso de herança múltipla.

Observe que não é preciso explicitar a chamada ao destrutor da classe base — a invocação da função é automática, sendo uma consequência da ligação entre as classes.

Quando o construtor da classe base requer argumentos, é preciso adotar a forma estendida de declaração de construtores. Esta forma pode ser representada como

```
construtor-derivado(lista-arg) : base(lista-arg) {  
    // código do construtor derivado  
    ...  
}
```

7.5 Outros Conceitos

Há em C++ muitos conceitos além destes apresentados aqui. Apenas para citar alguns, não se falou aqui sobre funções em linha, classes abstratas, funções virtuais, polimorfismo em tempo de execução, opções de controle de acesso a dados de classes derivadas, herança múltipla, *templates* ou sobre o tratamento de exceções. Espera-se no entanto que esta breve visão permita que se entenda porque se diz que C++ traz vantagens em relações a C e motive o leitor para um estudo posterior com mais calma.

7.6 Atividades

Atividade 1 Implemente e teste os exemplos do capítulo.

Apêndice A

Tabela ASCII

Apêndice B

Emacs

O ambiente AIX/RS6000 tem instalado, entre outros, o editor **emacs**. Este é um poderoso editor de tela, com recursos para trabalhar com várias janelas simultaneamente. Além de recursos incorporados em sua forma básica, **emacs** provê *modos* de edição específico para diferentes tipos de arquivos. **emacs** pode também ser configurado (através de uma linguagem própria) para suportar preferências individuais. A partir de **emacs** um usuário pode interagir com o shell em uma janela de edição, ler ou enviar mensagens através de e-mail, compilar um programa, e muito mais.

Sua forma básica de uso é

emacs *filename*

onde *filename* é o nome do arquivo a ser editado. A tela de **emacs** é composta por uma área de edição (onde uma ou mais janelas podem existir) e uma linha de comando. A cada janela de edição há uma linha de *status* associada que apresenta, entre outras informações, o nome do arquivo, o modo de edição, e uma indicação se o arquivo foi modificado ou não.

Para o usuário que não conhece os comandos do editor, um tutorial pode ser acessado através da sequência de comando CTRL-h t. Comandos em **emacs** são normalmente ativados através de uma sequência iniciada pelas teclas CTRL ou ESC. CTRL-h é uma sequências de acesso às diversas formas de *help*. Além de CTRL-h t, para tutorial, outra forma prática de auxílio é a sequência CTRL-h a, para *apropos*, que descreve todos os comandos relacionados a um dado *string*.

Entre os modos de **emacs**, há configurações para edição de programas em C, C++, e de arquivos L^AT_EX. Para cada modo, comandos adicionais podem ser suportados. Por exemplo, em modo C a tecla TAB não insere simplesmente um caracter de tabulação ou uma sequência de espaços, mas sim posiciona o início da linha corrente de acordo com o formato de indentação. Tais modos são ativados automaticamente quando o editor reconhece, pela extensão, qual o tipo de arquivo

sendo editado. Para obter quais os comandos associados a um modo, use a sequência CTRL-h m.

A seguir apresenta-se uma referência aos principais comandos de emacs.

Referência

Copyright © 1993 Free Software Foundation, Inc.
designed by Stephen Gildea, May 1993 v2.0
for GNU Emacs version 19 on Unix systems

Permission is granted to make and distribute copies of this card provided the copyright notice and this permission notice are preserved on all copies.

For copies of the GNU Emacs manual, write to the Free Software Foundation, Inc., 675 Massachusetts Ave, Cambridge MA 02139.

Starting Emacs

To enter GNU Emacs 19, just type its name: **emacs**

To read in a file to edit, see Files, below.

Leaving Emacs

suspend Emacs (or iconify it under X): C-z

exit Emacs permanently: C-x C-c

Files

read a file into Emacs: C-x C-f

save a file back to disk: C-x C-s

save **all** files: C-x s

insert contents of another file into this buffer: C-x i

replace this file with the file you really want: C-x C-v

write buffer to a specified file: C-x C-w

Getting Help

The Help system is simple. Type C-h and follow the directions.

If you are a first-time user, type C-h t for a **tutorial**.

remove Help window: C-x l

scroll Help window: ESC C-v

apropos: show commands matching a string: C-h a

show the function a key runs: C-h c
 describe a function: C-h f
 get mode-specific information: C-h m

Error Recovery

abort partially typed or executing command: C-g
recover a file lost by a system crash: M-x recover-file
undo an unwanted change: C-x u *or* C-_
 restore a buffer to its original contents: M-x revert-buffer
 redraw garbaged screen: C-l

Incremental Search

search forward: C-s
 search backward: C-r
 exit incremental search: RET
 undo effect of last character: DEL
 abort current search: C-g

Use C-s or C-r again to repeat the search in either direction.
 If Emacs is still searching, C-g cancels only the part not done.

Motion

character backward/forward: C-b / C-f
 word backward/forward: M-b / M-f
 line backward/forward: C-p / C-n
 go to line beginning/end: C-a / C-e
 go to buffer beginning/end: M-< / M->
 scroll to next/previous screen: C-v / M-v

Killing and Deleting

character back/forw (delete, not kill): DEL / C-d
 word back/forw: M-DEL / M-d
 line (to begin/end of): M-0 C-k / C-k
 kill **region**: C-w
 copy region to kill ring: M-w
 kill through next occurrence of *char*: M-z *char*
 yank back last thing killed: C-y

Marking

set mark here: C-@ *or* C-SPC
exchange point and mark: C-x C-x

Query Replace

interactively replace a text string: M-%
Valid responses in query-replace mode are:
replace this one, go on to next: SPC
replace this one, don't move: ,
skip to next without replacing: DEL
replace all remaining matches: !
exit query-replace: ESC

Multiple Windows

delete all other windows: C-x 1
delete this window: C-x 0
split window in two vertically: C-x 2
split window in two horizontally: C-x 3
scroll other window: C-M-v
switch cursor to another window: C-x o
select buffer in other window: C-x 4 b
find file in other window: C-x 4 f

Formatting

indent current **line** (mode-dependent): TAB
region (mode-dependent): C-M-\
insert newline after point: C-o
delete blank lines around point: C-x C-o
delete all white space around point: M-\
fill paragraph: M-q
set fill column: C-x f
set prefix each line starts with: C-x .

Case Change

uppercase word: M-u
lowercase word: M-l

capitalize word: M-c
 uppercase region: C-x C-u
 lowercase region: C-x C-l
 capitalize region: M-x capitalize-region

The Minibuffer

The following keys are defined in the minibuffer:

complete as much as possible: TAB
 complete up to one word: SPC
 complete and execute: RET
 show possible completions: ?
 fetch previous minibuffer input: M-p
 fetch next later minibuffer input: M-n
 regexp search backward through history: M-r
 regexp search forward through history: M-s
 abort command: C-g

Type C-x ESC ESC to edit and repeat the last command that used the minibuffer.

The following keys are then defined:

previous minibuffer command: M-p
 next minibuffer command: M-n

Buffers

select another buffer: C-x b
 list all buffers: C-x C-b
 kill a buffer: C-x k

Transposing

transpose **characters**: C-t
 transpose **words**: M-t
 transpose **lines**: C-x C-t

Info

enter the Info documentation reader: C-h i

Moving within a node:

scroll forward: SPC
 scroll reverse: DEL
 beginning of node: . (dot)

Moving between nodes:

next node: n
previous node: p
move **up**: u
select menu item by name: m
select *n*th menu item by number (1–5): n
follow cross reference (return with 1): f
return to last node you saw: l
return to directory node: d
go to any node by name: g

Other:

run Info **tutorial**: h
list Info commands: ?
quit Info: q
search nodes for regexp: s

Keyboard Macros

start defining a keyboard macro: C-x (
end keyboard macro definition: C-x)
execute last-defined keyboard macro: C-x e
append to last keyboard macro: C-u C-x (

Apêndice C

Compilador C

Para compilar um programa em C `fonte.c` gerando um arquivo executável `exec`, o comando a ser usado é da forma

```
$ cc fonte.c -o exec
```

Caso o nome do arquivo executável não seja fornecido (isto é, quando a opção “-o” for omitida), o compilador irá gerar automaticamente um programa de nome `a.out`.

Em alguns casos, é desejável compilar um programa fonte sem no entanto que um executável seja gerado. Para tais casos, a forma do comando a ser utilizada é

```
$ cc -c fonte.c
```

Com esta forma do comando, um arquivo objeto *fonte.o* será criado, e este arquivo poderá ser posteriormente ligado a outros módulos para gerar um programa executável.

Há também casos onde é necessário indicar ao compilador quais bibliotecas (além da biblioteca padrão da linguagem) contêm rotinas que estarão sendo usadas em seu programa. Há dois modos de indicar o uso destas bibliotecas. Quando a biblioteca é uma biblioteca do sistema (por exemplo, a biblioteca de rotinas de suporte a banco de dados `libdbm.a`), a forma de uso é

```
$ cc fonte1.c fonte2.c -o exec -ldb
```

ou seja, após a chave “-l” apenas o sufixo do nome da biblioteca é indicado. Esta linha de comando também ilustra o modo de se compilar diversos módulos em conjunto (neste caso, *fonte1.c* e *fonte2.c*). Se por outro lado a biblioteca for de uma aplicação que não seja padrão do sistema, então é suficiente indicar o nome (completo) da biblioteca na linha de comando, como em

```
$ cc fonte1.c fonte2.c -o exec mylib.a
```

Outras chaves que podem ser usadas frequentemente são `-D` (para definir um identificador como através de `#define`), `-I` (para indicar um caminho de busca para os arquivos de cabeçalho locais) e `-W` (para alterar o grau de verificação de possíveis erros — *warnings* — acusados pelo compilador).

O manual *online* (comandos `man` ou `xman`) contém a documentação completa sobre o comando `cc`. Sua Seção 3 contém informações sobre as rotinas da biblioteca C: além da lista de argumentos e valor de retorno para cada função, esta documentação também indica, quando necessário, que arquivos de cabeçalho (*headers*) devem ser incluídos.